PROCEEDINGS HCW '99

HETEROGENEOUS COMPUTING WORKSHOP

SAN JUAN, PUERTO RICO • APRIL 12, 1999

IEEE Computer Society Technical Committee on Parallel Processing
In cooperation with IPPS/SPDP
U.S. Office of Naval Research

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Sept. 9, 1999	Final, Nov. 1, 198 to Sept. 30, 1999

4. TITLE AND SUBTITLE
A 1999 Workshop on Heterogeneous Computing

5. FUNDING NUMBERS
N00014-99-1-0117

6. AUTHOR(S)
H. J. Siegel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Andre M. van Tilborg, Director
Math, Computer & Information Sciences Division
Office of Naval Research
Arlington, VA 22217-5660

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This grant funded the proceedings of the 8th Heterogeneous Computing Workshop (HCW '99), which was held on April 12, 1999. HCW '99 was part of the merged symposium of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP 1999), which was sponsored by the IEEE Computer Society Technical Committee on Parallel Processing and held in cooperation with ACM SIGARCH. Heterogeneous computing systems range from diverse elements within a single computer to coordinated, geographically distributed machines with different architectures. A heterogeneous computing system provides a variety of capabilities that can be orchestrated to execute multiple tasks with varied computational requirements. Applications in these environments achieve performance by exploiting the affinity of different tasks to different computational platforms or paradigms, while considering the overhead of inter-task communication and the coordination of distinct data sources and/or administrative domains. Topics representative of those in the proceedings include: network profiling, configuration tools, scheduling tools, analytic benchmarking, programming paradigms, problem mapping, processor assignment and scheduling, fault tolerance, programming tools, processor selection criteria, and compiler assistance.

14. SUBJECT TERMS
heterogeneous computing, distributed computing, high-performance computing

15. NUMBER OF PAGES
1

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

PURDUE UNIVERSITY

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
West Lafayette, Indiana 47907-1285, USA
E-mail: hj@purdue.edu

Prof. H. J. Siegel
Office Phone: 765-494-3444
Office Fax: 765-494-2706
Home Phone: 765-743-3290

September 9, 1999

Defense Technical Information Center
8725 John J. Kingman Road
STE 0944
Ft. Belvoir, Virginia 22060-6218

Enclosed is the final report (that is, form SF-298) for ONR grant number N00014-99-1-0117, which supported the publication of the enclosed workshop proceedings.

ONR's support is greatly appreciated.

Yours truly,

H. J. Siegel
Professor of
Electrical and Computer Engineering

cc: Dr. Andre van Tilborg, ONR
Grant Administrator, ONR
Purdue ECE Business Office

Proceedings

Eighth Heterogeneous Computing Workshop (HCW '99)


HETEROGENEOUS
COMPUTING
WORKSHOP

Proceedings

Eighth Heterogeneous Computing Workshop (HCW '99)

April 12, 1999
San Juan, Puerto Rico

Edited by
Viktor K. Prasanna, University of Southern California

Cosponsored by
IEEE Computer Society's Technical Committee on Parallel Processing
U.S. Office of Naval Research

Industrial Affiliate
NOEMIX

Los Alamitos, California
Washington • Brussels • Tokyo

IEEE Computer Society Order Number PR00107
ISBN 0-7695-0107-9
0-7695-0108-7 (microfiche)
0-7695-0109-5 (casebound)
ISSN 1097-5209

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1-714-821-8380
Fax: + 1-714-821-4641
E-mail: cs.books@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1-908-981-1393
Fax: + 1-908-981-9667
mis.custserv@computer.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-kuTokyo 107-0062
JAPAN
Tel: + 81-3-3408-3118
Fax: + 81-3-3408-3553
tokyo.ofc@computer.org

Editorial production by Lorretta Palagi

Cover art design and production by Alex Torres

Printed in the United States of America by Technical Communication Services

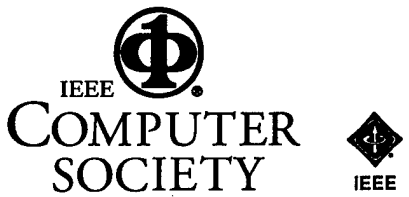


Table of Contents

Session IV: Task Assignment and Scheduling

Chair: Fusun Özgüner, The Ohio State University, Columbus, OH

Session V: Invited Case Studies

Chair: Noe Lopez-Benitez, Texas Tech University, Lubbock, TX, USA

Message from the General Chair

Welcome to the 8th Heterogeneous Computing Workshop. The field of heterogeneous computing is motivated by the diverse requirements of large-scale computational tasks, and the realization that the features of a single architecture are not always ideal for a wide range of task requirements.

HCW '99 is the result of the dedication and hard work of a number of people. I thank Richard F. Freund of NOEMIX for founding this series of workshops and for working hard to ensure its continuity and success. Special thanks also go to our industrial supporter, NOEMIX, for providing plaques of recognition to be awarded to individuals who have contributed to the workshop's success over the years.

Viktor K. Prasanna of the University of Southern California is this year's Program Chair. I have worked with Viktor on professional activities before, and as always he went above and beyond the call of duty. In addition to the Program Chair's tasks, he also took responsibility for completing tasks that I probably should have taken on in my role as General Chair. So I owe a special thank you to Viktor. With the able assistance of a terrific program committee, he has put together an excellent program and collection of papers in these proceedings.

Thanks are also due to the Steering Committee members for their guidance and support, and for the confidence they had in asking me to serve as this year's General Chair. Special thanks go to H. J. Siegel, the Steering Committee Chair, who did a remarkable job of leading that group and conveying ideas to me for enhancing the quality and prestige of the workshop. H. J. worked with Viktor and myself on numerous occasions with regard to planning and decision-making issues. H. J.'s advice, energy, and dedication to this workshop series are truly keys to its overall success.

The Publicity Chair, Muthucumaru Maheswaran of the University of Manitoba, did an outstanding job of publicizing the workshop through print and on the web. His careful and prompt updating of the workshop's web page was especially useful in keeping the authors informed on guidelines for final submission.

HCW '99 is being held in conjunction with IPPS/SPDP '99, the second merger of the International Parallel Processing Symposium (IPPS) and the Symposium on Parallel and Distributed Processing (SPDP). I thank the General Co-Chairs of IPPS/SPDP '99, Jose Rolim and Charles Weems, for their cooperation and assistance, with special thanks to Jose for taking on the responsibility of coordinating and organizing the workshops of IPPS/SPDP '99.

This year, the workshop is cosponsored by the IEEE Computer Society and the U.S. Office of Naval Research. These proceedings are published by the IEEE Computer Society Press. Deborah Plummer and Lorretta Palagi, both of the IEEE Computer Society Press, deserve special thanks for their punctuality and professionalism in overseeing the publication of these proceedings. Special thanks go to Lorretta for her efficient handling of the papers included here, and for carefully attending to the many details required to take a proceedings to press.

I would also like to thank my secretary, Marcelia Sawyers, for her assistance with my duties related to this workshop. Finally, I would like to thank my wife, Robin, for the loving support and patience she has for me.

John K. Antonio
Texas Tech University

Message from the Program Chair

The papers published in these proceedings represent some of the results from leading researchers in heterogeneous computing (HC). The field of heterogeneous computing has matured over the years. A number of experimental as well as commercial sytems continue to be built that integrate hardware, software, and algorithms to realize high-performance systems that satisfy diverse computational needs.

The response to the call for participation was excellent. Submissions were sent out to the program committee members for evaluation. In addition to their own reviews, the program committee members sought outside reviews to evaluate the submissions. The final selection of manuscripts was made at USC on November 24, 1998. The contributed papers were grouped into four sessions: Comparisons for Mapping Heuristics, Design Tools, Modeling and Analysis, and Task Assignment and Scheduling. In addition to contributed papers, the program includes a session of Invited Case Studies. I believe the papers represent continuing work as the field matures and I expect to see revised versions of these papers appear in archival journals.

I would like to thank many volunteers for their support. First of all, I want to thank John Antonio, General Chair, H. J. Siegel, Steering Committee Chair, and Richard Freund, who initiated the HCW series, for inviting me to be the program chair. Over the past year, John and H. J. provided me with a number of pointers to resolve meeting-related issues. I want to thank them for their invaluable inputs in composing a strong technical program. It was truly a pleasure working with them.

I would like to thank the authors for submitting their work and the program committee members and the reviewers for their efforts in reviewing the manuscripts. I would also like to thank Lorretta Palagi for her patience in working with late camera-ready submissions and for her prompt response to proceedings-related questions.

Finally, I am thankful to my assistant Henryk Chrostek who handled the submitted manuscripts in a timely manner.

Viktor K. Prasanna
University of Southern California

Message from the Steering Committee Chair

These are the proceedings of the 8th Heterogeneous Computing Workshop, also known as HCW '99. Heterogeneous computing is a very important research area with great practical impact. The topic of heterogeneous computing covers many types of systems. A heterogeneous system may be a set of machines interconnected by a wide-area network and used to support the execution of jobs submitted by a large variety of users to process data that is distributed throughout the system. A heterogeneous system may be a suite of high-performance machines tightly interconnected by a fast dedicated local-area network and used to process a set of production tasks, where the subtasks of each task may execute on different machines in the suite. A heterogeneous system may also be a special-purpose embedded system, such as a set of different types of processors used for automatic target recognition. In the extreme, a heterogeneous system may consist of a single machine that can reconfigure itself to operate in different ways (e.g., in different modes of parallelism). All of these types of heterogeneous systems (as well as others) are appropriate topics for this workshop series. I hope you find the contents of these proceedings informative and interesting, and encourage you to look also at the proceedings of past and future HCWs.

Many people have worked very hard to make this workshop happen. Viktor Prasanna, University of Southern California, is this year's Program Chair, and he assembled the great program that is represented by the papers in these proceedings. Viktor did this with the assistance of his Program Committee, which is listed on the next page. John Antonio, Texas Tech University, is the General Chair, and he is responsible for the overall organization and administration of this year's workshop, and he's done an outstanding job. I thank Richard F. Freund, NOEMIX, for founding this workshop series, and for asking me to succeed him as Chair of the Steering Committee.

This year the workshop is cosponsored by the IEEE Computer Society and the U.S. Office of Naval Research, with additional support from our industrial affiliate NOEMIX. I thank Andre M. van Tilborg, Director of the Math, Computer, & Information Sciences Division of the Office of Naval Research, for arranging funding for the publication of the workshop proceedings (under grant number N00014-99-1-0117). I thank Richard F. Freund, NOEMIX, for providing the plaque given to Viktor in recognition of his efforts as Program Chair.

This workshop is held in conjunction with the Merged International Parallel Processing Symposium & Symposium on Parallel and Distributed Processing (IPPS/SPDP). The HCW series is very appreciative of the constant cooperation and assistance we have received from the IPPS/SPDP organizers.

H. J. Siegel
School of Electrical and Computer Engineering
Purdue University

Organizing Committees

General Chair

John K. Antonio, *Texas Tech University*

Program Chair

Viktor K. Prasanna, *University of Southern California*

Steering Committee

H. J. Siegel, *Purdue University,* Chair
Francine Berman, *University of California at San Diego*
Jack Dongarra, *University of Tennessee and Oak Ridge National Lab*
Richard F. Freund, *NOEMIX*
Debra Hensgen, *Naval Postgraduate School*
Paul Messina, *Caltech*
Jerry Potter, *Kent State University*
Viktor K. Prasanna, *University of Southern California*
Vaidy Sunderam, *Emory University*

Publicity Chair

Muthucumaru Maheswaran, *University of Manitoba*

Program Committee

Gul A. Agha, *University of Illinois at Urbana-Champaign*
Ishfaq Ahmad, *The Hong Kong University of Science and Technology*
Francine Berman, *University of California at San Diego*
Hank Dietz, *Purdue University*
Jack Dongarra, *University of Tennessee and Oak Ridge National Lab*
Ian Foster, *Argonne National Laboratory*
Dennis Gannon, *Indiana University*
Andrew Grimshaw, *University of Virginia*
Babak Hamidzadeh, *University of British Columbia*
Salim Hariri, *University of Arizona*
Debra Hensgen, *Naval Postgraduate School*
Carl Kesselman, *ISI/University of Southern California*
Noe Lopez-Benitez, *Texas Tech University*
Muthucumaru Maheswaran, *University of Manitoba*
Veljko Milutinovic, *University of Belgrade*
Fusun Ozguner, *The Ohio State University*
Beth Plale, *Georgia Institute of Technology*
Cauligi Raghavendra, *The Aerospace Corporation*
Daniel A. Reed, *University of Illinois at Urbana-Champaign*
Jaspal Subhlok, *Carnegie Mellon University*
Rajeev Thakur, *Argonne National Laboratory*
Charles C. Weems, *University of Massachusetts*
Jon B. Weissman, *University of Texas at San Antonio*

Session I

Comparisons of Mapping Heuristics

Chair

Jon Weissman
University of Texas at San Antonio

Task Scheduling Algorithms for Heterogeneous Processors

Haluk Topcuoglu
Department of Electrical Engineering and Computer Science
Syracuse University, Syracuse, NY 13244-4100
haluk@top.cis.syr.edu

Salim Hariri
Department of Electrical and Computer Engineering
The University of Arizona, Tucson, Arizona 85721-0104

Min-You Wu
Department of Electrical and Computer Engineering
University of Central Florida

Abstract

Scheduling computation tasks on processors is the key issue for high-performance computing. Although a large number of scheduling heuristics have been presented in the literature, most of them target only homogeneous resources. The existing algorithms for heterogeneous domains are not generally efficient because of their high complexity and/or the quality of the results. We present two low-complexity efficient heuristics, the Heterogeneous Earliest-Finish-Time (HEFT) Algorithm and the Critical-Path-on-a-Processor (CPOP) Algorithm for scheduling directed acyclic weighted task graphs (DAGs) on a bounded number of heterogeneous processors. We compared the performances of these algorithms against three previously proposed heuristics. The comparison study showed that our algorithms outperform previous approaches in terms of performance (schedule length ratio and speedup) and cost (time-complexity).

1. Introduction

Efficient scheduling of application tasks is critical to achieving high performance in parallel and distributed systems. The objective of scheduling is to map the tasks onto the processors and order their execution so that task precedence requirements are satisfied and minimum schedule length (or *makespan*) is given. Since the general DAG scheduling is NP-complete, there are many research efforts that have proposed heuristics for the task scheduling problem [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].

Although a wide variety of different approaches are used to solve the DAG scheduling problem, most of them target only for homogeneous processors. The scheduling techniques that are suitable for homogeneous domains are limited and may not be suitable for heterogeneous domains. Only a few methods [5, 4, 9, 10] use variable execution times of tasks for heterogeneous environments; however, they are either high-complexity algorithms and/or they do not generally provide good quality of results.

In this paper we propose two static DAG scheduling algorithms for heterogeneous environments. They are for a bounded number of processors and are based on list-scheduling heuristics. The Heterogeneous Earliest-Finish-Time (HEFT) Algorithm selects the task with the highest upward rank (defined in Section 2) at each step; then the task is assigned to the most suitable processor that minimizes the earliest finish time with an insertion-based approach. The Critical-Path-on-a-Processor (CPOP) Algorithm schedules critical-path nodes onto a single processor that minimizes the critical path length. For the other nodes, the task selection phase of the algorithm is based on a summation of downward and upward ranks; the processor selection phase is based on the earliest execution finish time, as in the HEFT Algorithm. The simulation study in Section 5 shows that our algorithms considerably outperform previous approaches in terms of performance

0-7695-0107-9/99 $10.00

(schedule length ratio and speed-up) and cost (time-complexity).

The remainder of this paper is organized as follows. The next section gives the background of the scheduling problem, including some definitions and parameters used in the algorithms. In Section 3 we present the proposed scheduling algorithms for heterogeneous domains. Section 4 contains a brief review on the related scheduling algorithms that will be used in our comparison, and in Section 5 the performances of our algorithms are compared with the performances of related work, using task graphs of some real applications and randomly generated tasks graphs. Section 6 includes the conclusion and future work.

2. Problem Definition

A parallel/distributed application is decomposed into multiple tasks with data dependencies among them. In our model an application is represented by a directed acyclic graph (DAG) that consists of a tuple $G = (V, E, P, W, data, rate)$, where V is the set of v nodes/tasks, E is the set of e edges between the nodes, and P is the set of processors available in the system. (In this paper *task* and *node* terms are used interchangeably used.) Each edge $(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started.

W is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. Each $w_{i,j}$ gives the estimated execution time to complete task n_i on processor p_j. The average execution costs of tasks are used in the task priority equations. The average execution cost of a node n_i is defined as $\overline{w_i} = \sum_j^p w_{i,j}/p$. *Data* is a $v \times v$ matrix for data transfer size (in bytes) between the tasks. The data transfer rates (in bytes/second) between processors are stored in a $p \times p$ matrix, *rate*.

The communication cost of the edge (i, j), which is for data transfer from task n_i (scheduled on p_m) to task n_j (scheduled on p_n), is defined by $c_{i,j} = data(n_i, n_j)/rate(p_m, p_n)$. When both n_i and n_j are scheduled on the same processor, $p_m = p_n$, then $c_{i,j}$ becomes zero, since the intra-processor communication cost is negligible compared with the interprocessor communication cost. The average communication cost of an edge is defined by $\overline{c_{i,j}} = data(n_i, n_j)/\overline{rate}$, where $\overline{rate}$ is the average transfer rate between the processors in the domain.

The $EST(n_i, p_j)$ and $EFT(n_i, p_j)$ are the *earliest* execution start time and the *earliest* execution finish time of node n_i on processor p_j, respectively. They are defined by

$$EST(n_i, p_j) = \max\{\, T\_Available[j], \max_{n_m \in pred(n_i)} (EFT(n_m, p_k) + c_{m,i}) \,\} \quad (1)$$

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j) \quad (2)$$

where $pred(n_i)$ is the set of immediate predecessors of task n_i, and $T\_Available[j]$ is the earliest time at which processor p_j is available for task execution. The inner max block in the EST equation returns the *ready time*, i.e., the time when all data needed by n_i has arrived at the host p_j. The assignment decisions are stored in a two-dimensional matrix *list*. The j'th row of *list* matrix is for the sequence of nodes (in the order of execution start time) that was already scheduled on p_j.

The *objective function* of the scheduling problem is to determine an assignment of tasks of a given application to processors such that the *schedule length* (or *makespan*) is minimized by satisfying all precedence constraints. After all nodes in the DAG are scheduled, the schedule length will be the earliest finish time of the exit node n_e, $EFT(n_e, p_j)$, where exit node n_e is scheduled to processor p_j. (If a graph has multiple exit nodes, they are connected with zero-weight edges to a *pseudo exit node* that has zero computation cost. Similarly, a pseudo start node is added to the graphs with multiple start nodes).

The *critical path* (CP) of a DAG is the longest path from the entry node to the exit node in the graph. The length of this path, $|CP|$, is the sum of the computation costs of the nodes and inter-node communication costs along the path. The $|CP|$ value of a DAG is the lower bound of the schedule length.

In our algorithms we rank tasks as upward and downward to set the scheduling priorities. The *upward rank* of a task n_i is recursively defined by

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)) \quad (3)$$

where $succ(n_i)$ is the set of immediate successors of task n_i. Since it is computed recursively by traversing the task graph upward, starting from the exit node, it is referred to as an *upward* rank. Basically, $rank_u(n_i)$ is the length of the critical path (i.e., the longest path)

from n_i to the exit node, including the computation cost of the node itself. In some previous algorithms the ranks of the nodes are computed using computation costs only, which is referred to as *static* upward rank, $rank_u^s$.

Similarly, the *downward rank* of a task n_i is recursively defined by

$$rank_d(n_i) = \max_{n_j \in pred(n_i)} \{rank_d(n_j) + \overline{w_j} + \overline{c_{j,i}}\} \quad (4)$$

The downward ranks are computed recursively by traversing the task graph downward Basically, the $rank_d(n_i)$ is the longest distance from the start node to the node n_i, excluding the computation cost of the node itself.

In some previous algorithms the *level* attribute is used to set the priorities of the tasks. The level of a task is computed by the maximum number of edges along any path to the task from the start node. The start node has a level of zero.

3. Proposed Algorithms

In this section, we present our scheduling algorithms the Heterogeneous Earliest Finish Time (HEFT) Algorithm and the Critical-Path-on-a-Processor (CPOP) Algorithm.

3.1. The HEFT Algorithm

The Heterogeneous-Earliest-Finish-Time (HEFT) Algorithm, as shown in Figure 1, is a DAG scheduling algorithm that supports a bounded number of heterogeneous processing elements (PEs). To set priority to a task n_i, the HEFT algorithm uses the upward rank value of the task, $rank_u$ (Equation 3), which is the length of the longest path from n_i to the exit node. The $rank_u$ calculation is based on mean computation and communication costs. The task list is generated by sorting the nodes with respect to the decreasing order of the $rank_u$ values. In our implementation the ties are broken randomly; i.e., if two nodes to be scheduled have equal $rank_u$ values, one of them is selected randomly.

The HEFT algorithm uses the earliest finish time value, EFT, to select the processor for each task. In noninsertion-based scheduling algorithms, the earliest available time of a processor p_j, the $T\_Available[j]$ term in Equation 1, is the execution completion time of the last assigned node on p_j. The HEFT Algorithm, which is insertion-based, considers a possible insertion of each task in an earliest idle time slot between two already-scheduled tasks on the given processor. Formally, node n_i can be scheduled on processor p_j, which holds the following in equality for the minimum value of k

$$EST(list_{j,k+1}, p_j) - EFT(list_{j,k}, p_j) \geq w_{i,j} \quad (5)$$

where the $list_{j,k}$ is the kth node (in the start time sequence) that was already assigned on the processor p_j. Then, $T\_Available[j]$ will be equal to $EFT(list_{j,k}, p_j)$. The time complexity of the HEFT Algorithm is equal to $O(v^2 \times p)$.

3.2. The CPOP Algorithm

The Critical-Path-on-a-Processor (CPOP) Algorithm, shown in Figure 2, is another heuristic for scheduling tasks on a bounded number of heterogeneous processors. The $rank_u$ and $rank_d$ attributes of nodes are computed using mean computation and communication costs. The critical path nodes (CPN_i) are determined at Steps 5-6. The critical-path-processor (CPP) is the one that minimizes the length of the critical path (Step 7). The CPOP Algorithm uses $rank_d(n_i) + rank_u(n_i)$ to assign the node priority. The processor selection phase has two options: If the current node is on the critical path, it is assigned to the critical path processor (CPP); otherwise, it is assigned to the processor that minimizes the execution completion time. The latter option is insertion-based (as in the HEFT algorithm). At each iteration we maintain a priority queue to contain all free nodes and select the node that maximizes $rank_d(n_i) + rank_u(n_i)$. A binary heap was used to implement the priority queue, which has time complexity of $O(logv)$ for insertion and deletion of a node and $O(1)$ for retrieving the node with the highest priority (the root node of the heap). The time complexity of the algorithm is $O(v^2 \times p)$ for v nodes and p processors.

4. Related Work

Only a few of the proposed task scheduling heuristics support variable computation and communication costs for heterogeneous domains: the Dynamic Level Scheduling(DLS) Algorithm [4], the Levelized-Min Time (LMT) Algorithm [9], and the Mapping Heuristic (MH) Algorithm [5]. Although there are genetic algorithm based research efforts [6, 10, 14], most of them are slow and usually do not perform as well as the list-scheduling algorithms.

1. Compute $rank_u$ for all nodes by traversing graph upward, starting from the exit node.
3. Sort the nodes in a list by nonincreasing order of $rank_u$ values.
4. **while** there are unscheduled nodes in the list **do**
5. **begin**
6. Select the first task n_i in the list and remove it.
7. Assign the task n_i to the processor p_j that minimizes the (EFT) value of n_i.
8. **end**

Figure 1. The HEFT Algorithm

1. Compute $rank_u$ for all nodes by traversing graph upward, starting from the exit node.
2. Compute $rank_d$ for all nodes by traversing graph downward, starting from the start node.
3. $|CP| = rank_u(n_s)$, where n_s is the *start* node.
4. **For** each node n_i do
5. **If** $(rank_d(n_i) + rank_u(n_i) = |CP|)$ then
6. n_i is a critical path node (CPN).
7. Select the critical-path-processor that minimizes $\sum_{n_i \in CPN} w_{i,j}$.
8. Initialize the priority-queue with the entry nodes.
9. **while** there is an unscheduled node in the priority-queue do
10. **begin**
11. Select the highest priority node from priority-queue,
12. which maximizes $rank_d(n_i) + rank_u(n_i)$.
13. **if** (n_i is a CPN) then
14. schedule n_i to critical-path-processor.
15. **else**
16. Assign the task n_i to the processor p_j which minimizes the (EFT) value of n_i.
17. Update the priority-queue with the successor(s) of n_i if they become ready-nodes.
18. **end**

Figure 2. The CPOP Algorithm

Mapping Heuristic (MH) Algorithm The MH Algorithm uses *static* upward ranks ($rank_u^s$) to assign priorities to the nodes. A ready node list is kept sorted according to the decreasing order of priorities. (For tie-breaking, the node with the largest number of immediate successors is selected.) With a noninsertion-based method, the processor that provides the minimum earliest finish time of a task is selected to run the task. After a task is scheduled, the immediate successors of the task are inserted into the list. These steps are repeated until all nodes are scheduled. The time complexity of this algorithm is $O(v^2 \times p)$ for v nodes and p processors.

Dynamic-Level Scheduling (DLS) Algorithm The DLS Algorithm assigns node priorities by using an attribute called *Dynamic Level* (DL) that is equal to $DL(n_i, p_j) = rank_u(n_i) - EST(n_i, p_j)$. (In contrast to the mean values, *median* values are used to compute the static upward ranks; and for the EST computation, the noninsertion method is used). At each scheduling step the algorithm selects the (ready node, available processor) pair that maximizes the dynamic level value. For heterogeneous environments the $\Delta(n_i, p_j)$ term is added to the dynamic level computation. The Δ value for a task on a processor is computed by the difference between the task's median execution time on all processors and its execution time on the current processor. The DLS Algorithm has an $O(v^3 p \times f(p))$ time complexity, where v is the number of tasks and p is the number of processors. The complexity of the function used for data routing to calculate the earliest start time is $O(f(p))$ [4].

Levelized-Min Time Algorithm. This is a two-phase algorithm. The first phase orders the tasks based

on their precedence constraints, i.e., level by level. This phase groups the tasks that can be executed in parallel. The second phase is a greedy method that assigns each task (level by level) to the "fastest" available processor as much as possible. A task in a lower level has higher priority for scheduling than a node in a higher level; within the same level, the task with the highest average computation cost has the highest priority. If the number of tasks in a level is greater than the number of available processors, the fine-grain tasks are merged into a coarse-grain task until the number of tasks is equal to the number of processors. Then the tasks are sorted in reverse order (largest task first) based on average computation time. Beginning from the largest task, each task will be assigned to the processor: a) that minimizes the sum of computation cost of the task and the communication costs with tasks in the previous layers; and b) that does not have any scheduled task at the same level. For a fully-connected graph, the time complexity is $O(p^2v^2)$ when there are v tasks and p processors.

5. Performance and Comparison

In this section we present the performance comparison of our algorithms with the related work, using the randomly generated task graphs and regular task graphs representing the applications. The following metrics were used to compare the performances of our proposed algorithms with the previous approaches.

- **Schedule Length Ratio (SLR)**. The SLR of an algorithm is defined as
  $SLR = \frac{makespan}{\sum_{n_i \in CP_{MIN}} \min_{j \in P}\{w_{i,j}\}}$
  where *makespan* is the schedule length of the algorithm's output schedule. The denominator is the summation of computation costs of nodes on the CP_{MIN}. (For an unscheduled DAG, if the computation cost of each node n_i is set with $\{\min_{j \in P}\{w_{i,j}\}\}$, then the resulting critical path, CP_{MIN}, will be based on minimum computation values). The SLR value of any algorithm for a DAG cannot be less than one, since the denominator in the equation is the lower bound for completion time of the graph. The algorithm that gives the lowest SLR value of a DAG is the best algorithm with respect to performance (i.e., the one that gives the minimum overall execution time).

- **Speedup**. The speedup value of an algorithm is computed by dividing the overall computation time of the sequential execution to the makespan of the algorithm. (The sequential execution time is computed by assigning all tasks to a single processor that minimizes the total computation time of the DAG). Formally, it is defined as:
  $Speedup = \frac{min_{j \in P}(\sum_{n_i \in DAG} w_{i,j})}{makespan}$.

- **Running Time**. This is the average cost of each scheduling algorithm for obtaining the schedules of the given graphs on a Sun SPARC 10 workstation. The trade-offs between the performance (SLR value) and cost (running time) of scheduling algorithms were given in this section.

5.1. Generating Random Task Graphs

We developed an algorithm to generate random directed acyclic graphs that are used to evaluate the proposed scheduling algorithms. The input parameters required to generate a weighted random DAG are the following:

- Number of nodes (tasks) in the graph, v.

- Shape parameter of the graph, α. We assume that the height of a DAG is randomly generated from a uniform distribution with mean equal to $\alpha \times \sqrt{v}$. The width for each level in a DAG is randomly selected from a uniform distribution with mean equal to $\frac{\sqrt{v}}{\alpha}$. If $\alpha = 1.0$, then it will be a balanced DAG. A dense DAG (a shorter graph with high parallelism) can be generated by selecting $\alpha >> 1.0$. Similarly, if $\alpha << 1.0$, it will generate a longer DAG with a small parallelism degree.

- Out degree of a node, *out_degree*. One method is to use a fixed *out_degree* value for all nodes in a DAG as much as possible. Another alternative is to use a mean *out_degree*. Then, each node's out-degree will be randomly generated from a uniform distribution with mean equal to *out_degree*.

- Communication to computation ratio, CCR. CCR is the ratio of the average communication cost to the average computation cost. If a DAG's CCR value is low, it can be considered as a computation-intensive application; if it is high, it can be considered as a communication-intensive application.

- Average computation cost in the graph, *avg_comp*. Computation costs are generated randomly from a uniform distribution with mean equal to *avg_comp*. Similarly, the average communication cost is derived as $avg\_comm = CCR \times avg\_comp$.

- Range percentage of computation costs on processors, β. A high β value causes significant difference of a node's computation cost among the processors; a very low β value means that the expected execution times of a node on any given processors are almost equal. If the average computation cost of a node n_i is $\overline{w_i}$, then the computation cost of n_i on any processor p_j will be randomly selected from the range, $\overline{w_i} \times (1 - \frac{\beta}{2}) \leq w_{i,j} \leq \overline{w_i} \times (1 + \frac{\beta}{2})$, where β is the range percentage.
- Number of available processors in the system, *num_pe*.

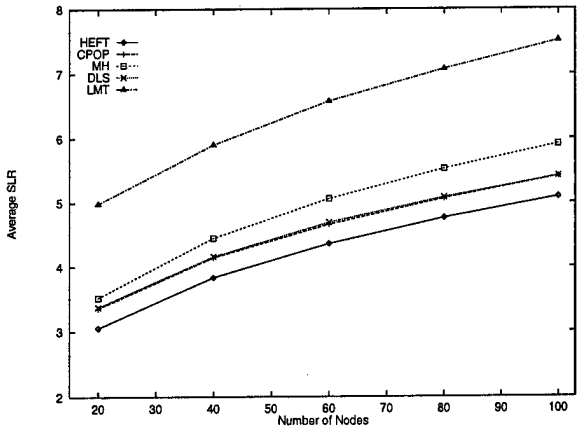
To compare the scheduling algorithms presented, we used random graphs as a test suite. The input parameters of the directed acyclic graph generation algorithm were set with the following values:

- $v = \{20, 40, 60, 80, 100\}$
- $CCR = \{0.1, 0.5, 1.0, 5.0, 10.0\}$
- $\alpha = \{0.5, 1.0, 2.0\}$
- *out_degree* $= \{1, 2, 3, 4, 5, 100\}$
- $\beta = \{0.1, 0.25, 0.5, 0.75, 1.0\}$

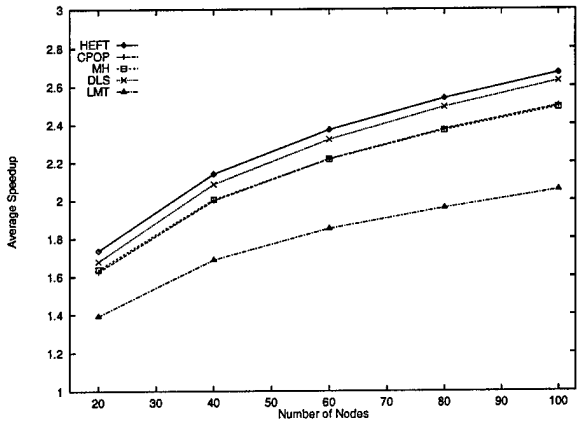
These combinations give 2250 different DAG types. Since we generated 25 random DAGs for each DAG type, the total number of DAGs used in our comparison study is around 56250. The following part gives the algorithm rankings for several graph parameters. Each ranking starts with the best algorithm and ends with the worst one, with respect to a given parameter.

Performance Study with Respect to Graph Size

The performances (SLR values) of algorithms were compared with respect to different graph sizes in Figure 3(a). The SLR value for each graph size is the average SLR value of 11250 different graphs that were generated randomly with different CCR, α, β, and *out_degree* values when the number of available processors was equal to 4. The performance ranking of the algorithms will be {HEFT, CPOP, DLS, MH, LMT }. The average SLR value of HEFT on all generated graphs will be better than CPOP by 7%, DLS Algorithm by 8%, MH Algorithm by 16% and LMT Algorithm by 52%. The average speedup curve with respect to the number of nodes is given in Figure 3(b). The average speedup ranking of the algorithms is {HEFT, DLS, (CPOP=MH), LMT}. We repeated these tests for the case when the number of processors was equal to 10. In this case, although the average SLR values were lower than in the previous case, they gave the same performance ranking of the algorithms.



(a)



(b)

Figure 3. (a) Average SLR (b) Average Speedup with Respect to Graph Size

Figure 4 shows the average running time of each algorithm. From this figure it can be concluded that the HEFT algorithm is the fastest and the DLS algorithm is slowest among the given algorithms. When the average running time of the algorithms on all graphs is computed by combining the results of different graph sizes, the HEFT Algorithm will be faster than the CPOP Algorithm by 10%, the MH Algorithm by 32%, the DLS Algorithm by 84%, and the LMT Algorithm by 48%.

Performance Study with Respect to Graph Structure

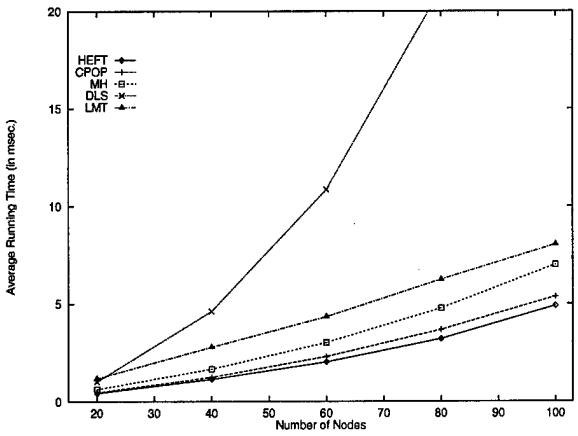
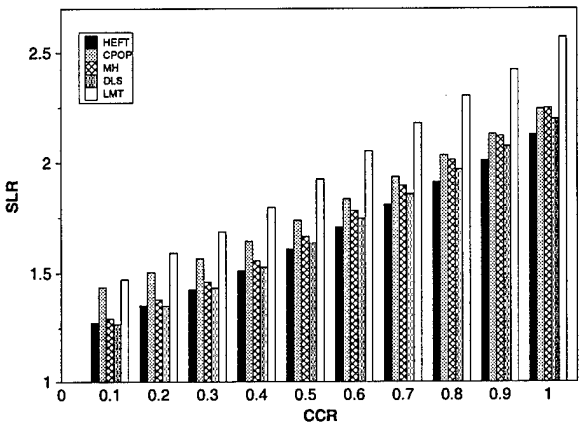


Figure 4. Average Running Time of the Algorithms with Respect to Graph Size

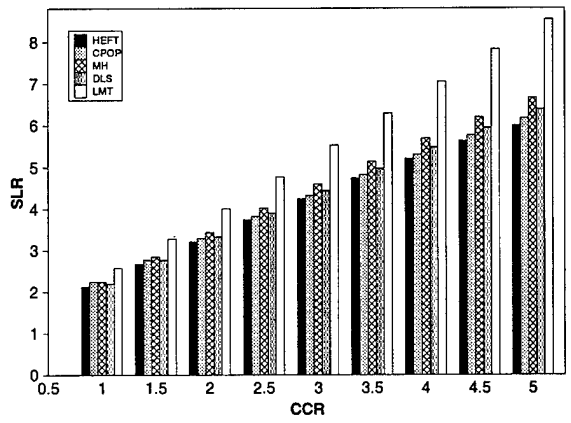
When α (the shape parameter of a graph) is equal to 0.5, the generated graphs have longer depths with a low parallelism degree. If the average SLR values of the algorithms are compared when $\alpha = 0.5$, the performance of the HEFT Algorithm is better than the CPOP Algorithm by 8%, the MH Algorithm by 12%, the DLS Algorithm by 6%, and the LMT Algorithm by 40%. When α is equal 1.0, the average depth of the graph and average width of each layer in the graph will be approximately equal. For this case, the average SLR value of the HEFT Algorithm will be better than the CPOP Algorithm by 7%, the MH Algorithm by 14%, the DLS Algorithm by 7%, and the LMT Algorithm by 34%. Similarly, if $\alpha = 2.0$, the average width will be approximately equal to four times the average depth, which will come up with short and dense graphs. For this case, the HEFT Algorithm will be better than the CPOP Algorithm by 6%, the MH Algorithm by 15%, the DLS Algorithm by 8%, and the LMT Algorithm by 31%. For all three α values, the HEFT Algorithm gives the best performance among the five algorithms.

Performance Study with Respect to CCR

Figure 5 gives the average SLR of the algorithms when the CCR value in $[0.1, 1.0]$ and $[1.0, 5.0]$ with number of processors is equal to 10. When $CCR \geq 0.25$, the HEFT Algorithm outperforms the other algorithms. If $CCR \leq 0.2$, the DLS algorithm gives a better performance than the HEFT Algorithm. The performance ranking of the algorithms (when $CCR \leq 1.0$) is {HEFT, DLS, MH, CPOP, LMT}. When $CCR > 1.0$ the performance ranking changes to {HEFT, CPOP, DLS, MH, LMT}.



(a)



(b)

Figure 5. Average SLR of the Algorithms with Respect to CCR: (a) $0.1 \leq CCR \leq 1.0$ **(b)** $1.0 \leq CCR \leq 10.0$

5.2. Applications

In this section we present a performance comparison of the scheduling algorithms based on Gaussian elimination and Fast Fourier Transformation (FFT). For the Gaussian elimination algorithm, we did the analytical work to set approximately the computation costs of the nodes and communication costs of the edges. The number of nodes in the Gaussian elimination algorithm can be characterized by the input matrix size. For FFT, the computation cost of each node is randomly set from a normal distribution with mean equal to an assigned average computation cost. The communication costs are set randomly from a normal distribution with mean equal to the multiplication of CCR and average communication cost. The number of nodes in an FFT task graph can be computed in terms of number of input points (i.e., order of the FFT).

In order to provide different processor speeds, the *computing power weights* [18] of the processors were randomly set from a given range. The *computing power weight* of each processor P_j, (φ^j), is a number that shows the ratio of CPU speed of P_j to the CPU speed of the fastest processor in the system. In order to set the computation cost of each task on processors, the computation cost of the node on a base processor (P_{base}) is set (either randomly as in FFT or analytically as in the Gaussian elimination). Then the computation cost for each other processor P_i is computed by $W(T_j, P_k) = \frac{\varphi^{base}}{\varphi^k} \times W(T_j, P_{base})$. For the fastest processor, P_F, $\varphi^F = 1$.

In the experiments the range of the computing power weights can be $[0.8-1.0]$, $[0.6-1.0]$, $[0,4-1.0]$ or $[0.2-1.0]$. The first range is for a domain in which the computational powers of processors are almost equal; however, at the last set ($[0.2-1.0]$) the fastest processor can be up to five times faster than the slowest one. We also varied the task graph granularities by varying the CCR values in the range $\{0.1, 0.5, 1.0, 2.0, 5.0, 10.0\}$.

Gaussian elimination. Figure 6(a) gives the sequential program for the Gaussian elimination algorithm [1, 17]. The data-flow graph of the algorithm for the special case of $n = 5$, where n is the dimension of the matrix, is given in Figure 6(b). The number of nodes in the task graph of this algorithm is roughly $\frac{n^2+n-2}{2}$. Each $T_{k,k}$ represents a pivot column operation, and each $T_{k,j}$ represents an update operation. The node computation cost function for any task $T_{k,j}$ is equal to $W_{k,j} = 2 \times (n-k) \times w$, where $1 \leq k \leq j \leq n$ (w is the average execution time of either an addition and multiplication, or of a division).

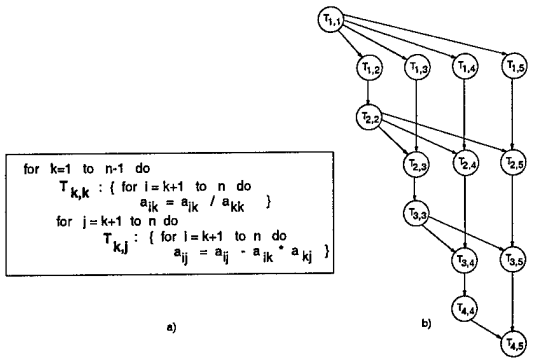
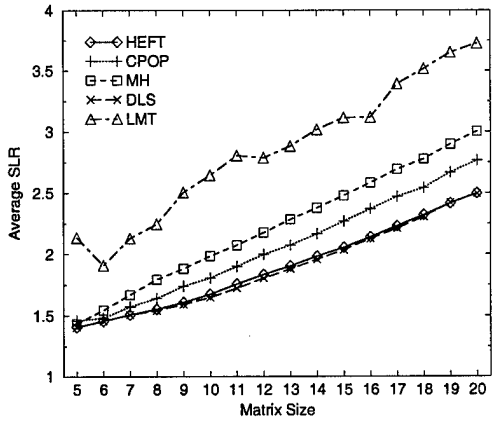


Figure 6. (a) The Gaussian elimination Algorithm (kji version) (b) The Task graph for a Matrix of Size 5

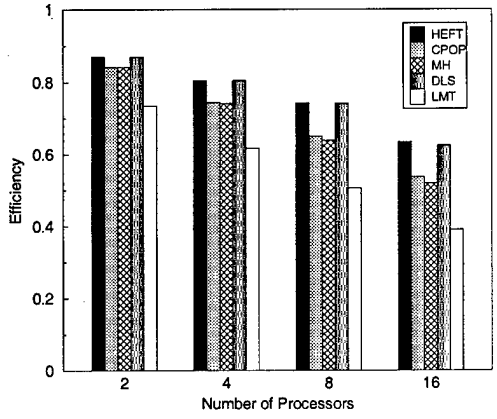
The edge communication cost function between any edge is $C_{k,j} = (n-k) \times b$, where b is the transmission rate. Each Gaussian elimination task graph has one critical path (CP), which has the maximum number of nodes covered as compared to the other paths. In Figure 6(b), the critical path is $T_{1,1}T_{1,2}T_{2,2}T_{2,3}T_{3,3}T_{3,4}T_{4,4}T_{4,5}$.

Figure 7(a) gives the average SLR values of the algorithms at various matrix sizes. The performances of the HEFT and DLS algorithms were the best of all. The performance ranking of the algorithms for Gaussian elimination graphs is {(HEFT = DLS), CPOP, MH, LMT}. As in the results of other applications, the SLR values of all scheduling algorithms slightly increase if the matrix size is increased. Increasing the matrix size causes more nodes not to be on the critical path, which causes increases in the makespan.

Figure 7(b) gives the efficiency comparison of the algorithms for the Gaussian elimination graphs of about 1250 nodes (i.e., the matrix size is 50) at various numbers of processors. Efficiency is the ratio of the speedup value to the number of processors used. The HEFT and DLS algorithms have greater efficiency than the other algorithms; when the number of processors is increased beyond eight, the HEFT algorithm outperforms the DLS algorithm in terms of efficiency. For all scheduling algorithms, increasing the number of processors reduces efficiency because the matrix size (thus the number of nodes) is fixed. Additionally, an increase in the number of processors causes fewer nodes than the number of processors at some levels of the graph, which will decrease the utilization of the processors.



(a)



(b)

Figure 7. (a) Average SLRs of Scheduling Algorithms at Various Graph Sizes for Gaussian Elimination Graph (b) The Efficiency Comparison of the Algorithms

Figure 8 gives the average running time of each algorithm for the Gaussian elimination graph for a various number of processors. Although the DLS algorithm gives as good performance as the HEFT algorithm for Gaussian elimination graphs, it is the slowest algorithm. (For Gaussian elimination task graphs when matrix size is 50 with 16 processors, the DLS algorithm takes 16.2 times longer time than the HEFT algorithm to schedule tasks). The running time of the HEFT algorithm is comparable to the MH algorithm, but it is greater than the LMT algorithm. The cost ranking of the algorithms (starting from the lowest) for the Gaussian elimination graphs is {LMT,(HEFT, MH),CPOP,DLS}. Although the LMT Algorithm is a higher-complexity algorithm than the others except for the DLS algorithm (see Figure 4), it gives the best running time for Gaussian elimination graphs due to the fact that half of the levels of a Gaussian elimination graph has a single node, which decreases the cost of the LMT algorithm.

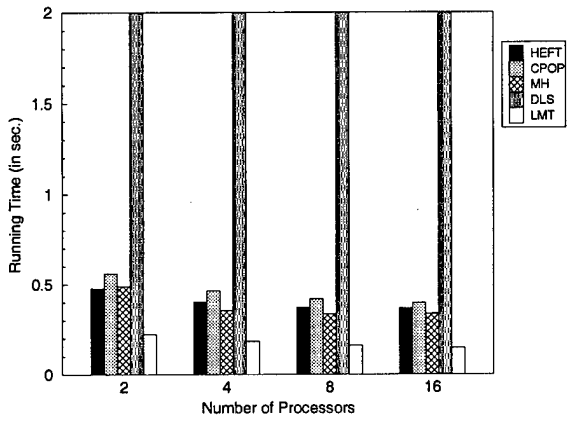


Figure 8. Average Running Time for Each Algorithm to Schedule Gaussian Elimination Graphs

Fast Fourier Transformation The recursive, one-dimensional FFT Algorithm [15, 16] and its task graph is given in Figure 9. A is an array of size n, which holds the coefficients of the polynomial; array Y is the output of the algorithm. The algorithm consists of two parts: recursive calls (lines 3-4) and the butterfly operation (lines 6-7). The task graph in Figure 9(b) can be divided into two parts: the nodes above the dashed line are the recursive call nodes (RCNs), and the ones below the line are butterfly operation nodes (BONs). For an input vector of size n, there are $2 \times n - 1$ RCNs and

$n \times \log_2 n$ BONs. (We assume that $n = 2^m$ for some integer m). Each path from the start node to any of the exit nodes in an FFT task graph is a critical path because the computation cost of nodes in any level are equal, and the communication costs of edges between two consecutive levels are equal.

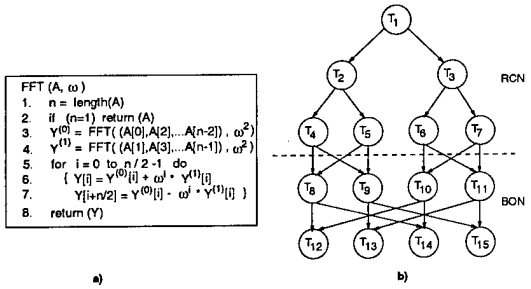
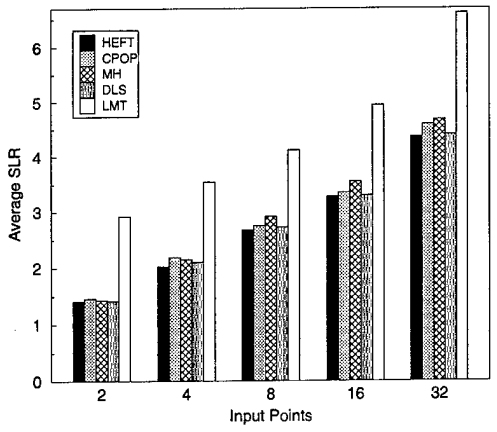
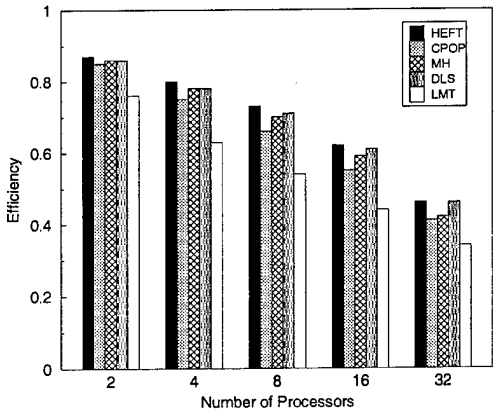


Figure 9. (a) FFT Algorithm (b) The Generated DAG of FFT with Four Points.

Figure 10(a) shows the average SLR values of scheduling algorithms for FFT graphs for various numbers of input points when the number of processors is equal to six. The HEFT algorithm outperforms the other algorithms. The performance ranking of the algorithms is {HEFT, DLS, CPOP, MH, LMT}. Figure 10(b) shows the efficiency values obtained for each of the algorithms when there are 64 data points. The HEFT and DLS algorithms result in good schedules for all cases. The running time comparisons of the algorithms, with respect to number of input points and with respect to number of processors, were shown in Figure 11. For FFT graphs, the DLS and LMT Algorithms are high-cost scheduling algorithms, whereas the HEFT, CPOP, and MH Algorithms are low-complexity scheduling algorithms. Based on these results it can be concluded that the HEFT algorithm is the best algorithm in terms of performance and cost.



(a)

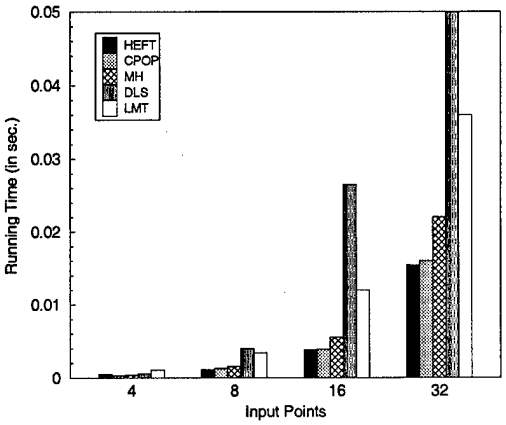


(b)

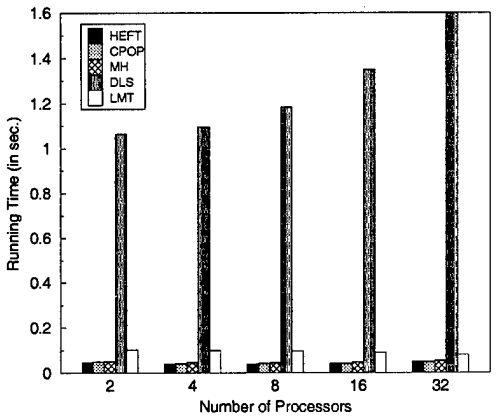
Figure 10. (a) Average SLRs of Scheduling Algorithms at Various Graph Sizes for FFT Graph (b) Efficiency Comparison of the Algorithms

6. Conclusion and Future Work

In this paper we have proposed two task scheduling algorithms (the HEFT Algorithm and the CPOP Algorithm) for heterogeneous processors. For the generated random task graphs and the task graphs of for selected applications, the HEFT algorithm outperforms the other algorithms in all performance metrics, i.e., average schedule length ratio (SLR), speedup, and time-complexity. Similarly, the CPOP Algorithm is better than, or at least comparable to, the existing algorithms. Both algorithms perform more stably than



(a)



(b)

Figure 11. Running Times of the Scheduling Algorithms for FFT Graphs

the others in terms of scheduling quality and running time.

We are extending our algorithms to improve their performance at specific CCR ranges and computing power weight ranges. Additionally, we plan to add low-cost, local-search techniques to improve the scheduling quality of our algorithms. Future work will include different techniques for ordering the ready tasks and extensions for the processor selection phase.

References

[1] M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 330–343, July 1990.

[2] Y. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, May 1996.

[3] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 113–120, Feb. 1994.

[4] G. C. Sih and E. A. Lee, A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 175–186, Feb. 1993.

[5] H. El-Rewini and T. G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 138–153, 1990.

[6] H. Singh, A. Youssef, "Mapping and Scheduling Heterogeneous Task Graphs Using Genetic Algorithms," *Heterogeneous Computing Workshop*, 1996.

[7] I. Ahmad and Y. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," *Proc. of Int'l Conference on Parallel Processing*, vol. II, pp. 47–51, 1994.

[8] M. A. Palis, J. Liou, and D. S. L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, 46–55, Jan. 1996.

[9] M. Iverson, F. Ozguner, G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environments," *Proc. of Heterogeneous Computing Workshop*, 93–100, 1995.

[10] P. Shroff, D. W. Watson, N. S. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," *Proc. of Heterogeneous Computing Workshop*, 1996.

[11] T. Yang and A. Gerasoulis, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, 276–291, 1992.

[12] T. Yang and A. Gerasoulis. "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, Sept. 1994.

[13] Y. Kwok and I. Ahmad, "A Static Scheduling Algorithm Using Dynamic Critical Path For Assigning Parallel Algorithms onto Multiprocessors," *Proc. of Int'l Conference on Parallel Processing*, vol. II, pp. 155–159, 1994.

[14] L. Wang, H. J. Siegel, and V. P. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments," *Proc. of Heterogeneous Computing Workshop*, 1996.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[16] Y. Chung and S. Ranka, "Applications and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors," *Proc. of Supercomputing*, pp. 512–521, Nov. 1992.

[17] M. Cosnard, M. Marrakchi, Y. Robert, D. Trystram, "Parallel Gaussian Elimination on an MIMD Computer," *Parallel Computing*, vol. 6, pp. 275-295, 1988.

[18] Y. Yan, X. Zhang, Y. Song, "An Effective Performance Prediction Model for Parallel Computing on Non-dedicated Heterogeneous NOW," *Journal of Parallel and Distributed Computing*, vol. 38, pp. 63-80, 1996.

Biographies

Haluk Topcuoglu is a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at Syracuse University, New York, USA. His research interests include task scheduling techniques in heterogeneous environments, metacomputing and cluster computing issues, parallel and distributed programming, and web technologies. He received his B.S. and M.S. degrees in 1991 and 1993, respectively, in computer engineering from Bogazici University, Istanbul, Turkey. Mr. Topcuoglu is a member of ACM, IEEE, IEEE Computer Society, and the Phi Beta Delta Honor Society.

Salim Hariri is currently a Professor in the Department of Electrical and Computer Engineering at The University of Arizona. Dr. Hariri received his Ph.D in computer engineering from University of Southern California in 1986, an M.S. degree from The Ohio State University, in 1982, and B.S. in Electrical Engineering from Damascus University, in 1977. His current research focuses on high performance distributed computing, design and analysis of high speed networks, benchmarking and evaluating parallel and distributed systems, and developing software design tools for high performance computing and communication systems and applications. Dr. Hariri is the Editor-In-Chief for the Cluster Computing Journal. He served as the Program Chair and the General Chair of the IEEE International Symposium on High Performance Distributed Computing (HPDC).

Min-You Wu received the M.S. degree from the Graduate School of Academia Sinica, Beijing, China, and the Ph.D. degree from Santa Clara University, California. Before he joined the Department of Electrical and Computer Engineering, University of Central Florida, where he is currently an Associate Professor, he has held various positions at University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, and State University of New York at Buffalo. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 70 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a member of ACM and a senior member of IEEE. He is listed in International Who's Who of Information Technology.

A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems

Tracy D. Braun[†], Howard Jay Siegel[†], Noah Beck[†], Ladislau L. Bölöni[‡], Muthucumaru Maheswaran[§], Albert I. Reuther[†], James P. Robertson[*], Mitchell D. Theys[†], Bin Yao[†], Debra Hensgen[◇], and Richard F. Freund[¶]

[†]School of Electrical and Computer Engineering
[‡]Department of Computer Sciences
Purdue University
West Lafayette, IN 47907 USA
{tdbraun, hj, noah, reuther, theys, yaob}
@ecn.purdue.edu, boloni@cs.purdue.edu

[§]Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2 Canada
maheswar@cs.umanitoba.ca

[*]Motorola
6300 Bridgepoint Parkway
Bldg. #3, MD: OE71
Austin, TX 78730 USA
robertso@ibmoto.com

[◇]Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5118 USA
hensgen@cs.nps.navy.mil

[¶]NOEMIX
1425 Russ Blvd., Ste. T-110
San Diego, CA 92101 USA
rffreund@noemix.com

Abstract

Heterogeneous computing (HC) environments are well suited to meet the computational demands of large, diverse groups of tasks (i.e., a meta-task). The problem of mapping (defined as matching and scheduling) these tasks onto the machines of an HC environment has been shown, in general, to be NP-complete, requiring the development of heuristic techniques. Selecting the best heuristic to use in a given environment, however, remains a difficult problem, because comparisons are often clouded by different underlying assumptions in the original studies of each heuristic. Therefore, a collection of eleven heuristics from the literature has been selected, implemented, and analyzed under one set of common assumptions. The eleven heuristics examined are Opportunistic Load Balancing, User-Directed Assignment, Fast Greedy, Min-min, Max-min, Greedy, Genetic Algorithm, Simulated Annealing, Genetic Simulated Annealing, Tabu, and A. This study provides one even basis for comparison and insights into circumstances where one technique will outperform another. The evaluation procedure is specified, the heuristics are defined, and then selected results are compared.*

This research was supported in part by the DARPA/ITO Quorum Program under NPS subcontract numbers N62271-98-M-0217 and N62271-98-M-0448. Some of the equipment used was donated by Intel.

1. Introduction

Mixed-machine heterogeneous computing (HC) environments utilize a distributed suite of different high-performance machines, interconnected with high-speed links to execute different computationally intensive applications that have diverse computational requirements [10, 18, 24]. The general problem of mapping (i.e., matching and scheduling) tasks to machines has been shown to be NP-complete [8, 15]. Heuristics developed to perform this mapping function are often difficult to compare because of different underlying assumptions in the original studies of each heuristic [3]. Therefore, a collection of eleven heuristics from the literature has been selected, implemented, and compared by simulation studies under one set of common assumptions.

To facilitate these comparisons, certain simplifying assumptions were made. Let a meta-task be defined as a collection of independent tasks with no data dependencies (a given task, however, may have subtasks and dependencies among the subtasks). For this case study, it is assumed that static (i.e., off-line or predictive) mapping of meta-tasks is being performed. (In some systems, all tasks and subtasks in a meta-task, as defined above, are referred to as just tasks.)

It is also assumed that each machine executes a single task at a time, in the order in which the tasks ar-

0-7695-0107-9/99 $10.00 © 1999 IEEE

rived. Because there are no dependencies among the tasks, scheduling is simplified, and thus the resulting solutions of the mapping heuristics focus more on finding an efficient matching of tasks to machines. It is also assumed that the size of the meta-task (number of tasks to execute), t, and the number of machines in the HC environment, m, are static and known *a priori*.

Section 2 defines the computational environment parameters that were varied in the simulations. Descriptions of the eleven mapping heuristics are found in Section 3. Section 4 examines selected results from the simulation study. A list of implementation parameters and procedures that could be varied for each heuristic is presented in Section 5.

This research was supported in part by the DARPA/ITO Quorum Program project called MSHN (Management System for Heterogeneous Networks) [13]. MSHN is a collaborative research effort among the Naval Postgraduate School, NOEMIX, Purdue University, and the University of Southern California. The technical objective of the MSHN project is to design, prototype, and refine a distributed resource management system that leverages the heterogeneity of resources and tasks to deliver requested qualities of service. The heuristics developed in this paper or their derivatives may be included in the Scheduling Advisor component of the MSHN prototype.

2. Simulation Model

The eleven static mapping heuristics were evaluated using simulated execution times for an HC environment. Because these are static heuristics, it is assumed that an accurate estimate of the expected execution time for each task on each machine is known prior to execution and contained within an *ETC* (expected time to compute) matrix. One row of the *ETC* matrix contains the estimated execution times for a given task on each machine. Similarly, one column of the *ETC* matrix consists of the estimated execution times of a given machine for each task in the meta-task. Thus, $ETC(i,j)$ is the estimated execution time for task i on machine j. (These times are assumed to include the time to move the executables and data associated with each task to the particular machine when necessary.) The assumption that these estimated expected execution times are known is commonly made when studying mapping heuristics for HC systems (e.g., [11, 16, 25]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [14, 24].)

For the simulation studies, characteristics of the *ETC* matrices were varied in an attempt to represent a variety of possible HC environments. The *ETC* matrices used were generated using the following method. Initially, a $t \times 1$ baseline column vector, B, of floating point values is created. Let ϕ_b be the upper-bound of the range of possible values within the baseline vector. The baseline column vector is generated by repeatedly selecting a uniform random number, $x_b^i \in [1,\ \phi_b)$, and letting $B(i) = x_b^i$ for $0 \leq i < t$. Next, the rows of the *ETC* matrix are constructed. Each element $ETC(i,j)$ in row i of the *ETC* matrix is created by taking the baseline value, $B(i)$, and multiplying it by a uniform random number, $x_r^{i,j}$, which has an upper-bound of ϕ_r. This new random number, $x_r^{i,j} \in [1,\ \phi_r)$, is called a row multiplier. One row requires m different row multipliers, $0 \leq j < m$. Each row i of the *ETC* matrix can be then described as $ETC(i,j) = B(i) \times x_r^{i,j}$, for $0 \leq j < m$. (The baseline column itself does not appear in the final *ETC* matrix.) This process is repeated for each row until the $m \times t$ *ETC* matrix is full. Therefore, any given value in the *ETC* matrix is within the range $[1,\ \phi_b \times \phi_r)$.

To evaluate the heuristics for different mapping scenarios, the characteristics of the *ETC* matrix were varied based on several different methods from [2]. The amount of variance among the execution times of tasks in the meta-task for a given machine is defined as task heterogeneity. Task heterogeneity was varied by changing the upper-bound of the random numbers within the baseline column vector. High task heterogeneity was represented by $\phi_b = 3000$ and low task heterogeneity used $\phi_b = 100$. Machine heterogeneity represents the variation that is possible among the execution times for a given task across all the machines. Machine heterogeneity was varied by changing the upper-bound of the random numbers used to multiply the baseline values. High machine heterogeneity values were generated using $\phi_r = 1000$, while low machine heterogeneity values used $\phi_r = 10$. These heterogeneous ranges are based on one type of expected environment for MSHN.

To further vary the *ETC* matrix in an attempt to capture more aspects of realistic mapping situations, different *ETC* matrix consistences were used. An *ETC* matrix is said to be consistent if whenever a machine j executes any task i faster than machine k, then machine j executes all tasks faster than machine k [2]. Consistent matrices were generated by sorting each row of the *ETC* matrix independently. In contrast, inconsistent matrices characterize the situation where machine j is faster than machine k for some tasks, and slower for others. These matrices are left in the unordered, random state in which they were generated. In between these two extremes, semi-consistent matrices represent a partial ordering among the machine/task

execution times. For the semi-consistent matrices used here, the row elements in column positions $\{0, 2, 4, \ldots\}$ of row i are extracted, sorted, and replaced in order, while the row elements in column positions $\{1, 3, 5, \ldots\}$ remain unordered. (That is, the even columns are consistent and the odd columns are inconsistent.)

Sample ETC matrices for the four inconsistent heterogeneous permutations of the characteristics listed above are shown in Tables 1 through 4. (Other probability distributions for ETC values, including an exponential distribution and a truncated Gaussian [1] distribution, were also investigated, but not included in the results discussed here.) All results in this study used ETC matrices that were of size $t = 512$ tasks by $m = 16$ machines. While it was necessary to select some specific parameter values to allow implementation of a simulation, the characteristics and techniques presented here are completely general. Therefore, if these parameter values do not apply to a specific situation of interest, researchers may substitute in their own ranges, distributions, matrix sizes, etc., and the evaluation software of this study will still apply.

3. Heuristic Descriptions

The definitions of the eleven static meta-task mapping heuristics are provided below. First, some preliminary terms must be defined. Machine availability time, $mat(j)$, is the earliest time a machine j can complete the execution of all the tasks that have previously been assigned to it. Completion time, $ct(i, j)$, is the machine availability time plus the execution time of task i on machine j, i.e., $ct(i, j) = mat(j) + ETC(i, j)$. The performance criterion used to compare the results of the heuristics is the maximum value of $ct(i, j)$, for $0 \leq i < t$ and $0 \leq j < m$, for each mapping, also known as the makespan [19]. Each heuristic is attempting to minimize the makespan (i.e., finish execution of the meta-task as soon as possible).

The descriptions below implicitly assume that the machine availability times are updated after each task is mapped. For cases when tasks can be considered in an arbitrary order, the order in which the tasks appeared in the ETC matrix was used. Some of the heuristics listed below had to be modified from their original implementation to better handle the scenarios under consideration.

For many of the heuristics, there are control parameter values and/or control function specifications that can be selected for a given implementation. For the studies here, such values and specifications were selected based on experimentation and/or information in the literature. These parameters and functions are mentioned in Section 5.

OLB: Opportunistic Load Balancing (OLB) assigns each task, in arbitrary order, to the next available machine, regardless of the task's expected execution time on that machine [1, 9, 10].

UDA: In contrast to OLB, User-Directed Assignment (UDA) assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability [1]. UDA is sometimes referred to as Limited Best Assignment (LBA), as in [1, 9].

Fast Greedy: Fast Greedy assigns each task, in arbitrary order, to the machine with the minimum completion time for that task [1].

Min-min: The Min-min heuristic begins with the set U of all unmapped tasks. Then, the set of minimum completion times, $M = \{m_i : m_i = \min_{0 \leq j < m}(ct(i, j))$, for each $i \in U\}$, is found. Next, the task with the overall *minimum* completion time from M is selected and assigned to the corresponding machine (hence the name Min-min). Lastly, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., $U = \emptyset$) [1, 9, 15].

Intuitively, Min-min attempts to map as many tasks as possible to their first choice of machine (on the basis of completion time), under the assumption that this will result in a shorter makespan. Because tasks with shorter execution times are being mapped first, it was expected that the percentage of tasks that receive their first choice of machine would generally be higher for Min-min than for Max-min (defined next), and this was verified by data recorded during the simulations.

Max-min: The Max-min heuristic is very similar to Min-min. The Max-min heuristic also begins with the set U of all unmapped tasks. Then, the set of minimum completion times, $M = \{m_i : m_i = \min_{0 \leq j < m}(ct(i, j))$, for each $i \in U\}$, is found. Next, the task with the overall *maximum* completion time from M is selected and assigned to the corresponding machine (hence the name Max-min). Lastly, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e., $U = \emptyset$) [1, 9, 15].

The motivation behind Max-min is to attempt to minimize the penalties incurred by delaying the scheduling of long-running tasks. Assume that the meta-task being mapped has several tasks with short execution times, and a small quantity of tasks with very long execution times. Mapping the tasks with the longer execution times to their best machines first allows these tasks to be executed concurrently with the remaining tasks (with shorter execution times). This concurrent execution of long and short tasks can be more beneficial than a Min-min mapping where all of

the shorter tasks would execute first, and then a few longer running tasks execute while several machines sit idle. The assumption here is that with Max-min the tasks with shorter execution times can be mixed with longer tasks and evenly distributed among the machines, resulting in better machine utilization and a better meta-task makespan.

Greedy: The Greedy heuristic is literally a combination of the Min-min and Max-min heuristics. The Greedy heuristic performs both of the Min-min and Max-min heuristics, and uses the better solution [1, 9].

GA: Genetic Algorithms (GAs) are a popular technique used for searching large solution spaces (e.g., [25, 27]). The version of the heuristic used for this study was adapted from [27] for this particular solution space. Figure 1 shows the steps in a general Genetic Algorithm.

The Genetic Algorithm implemented here operates on a population of 200 chromosomes (possible mappings) for a given meta-task. Each chromosome is a $t \times 1$ vector, where position i $(0 \leq i < t)$ represents task i, and the entry in position i is the machine to which the task has been mapped. The initial population is generated using two methods: (a) 200 randomly generated chromosomes from a uniform distribution, or (b) one chromosome that is the Min-min solution and 199 random solutions (mappings). The latter method is called seeding the population with a Min-min chromosome. The GA actually executes eight times (four times with initial populations from each method), and the best of the eight mappings is used as the final solution.

After the generation of the initial population, all of the chromosomes in the population are evaluated (i.e., ranked) based on their fitness value (i.e., makespan), with a smaller fitness value being a better mapping. Then, the main loop in Figure 1 is entered and a rank-based roulette wheel scheme [26] is used for selection. This scheme probabilistically generates new populations, where better mappings have a higher probability of surviving to the next generation. Elitism, the property of guaranteeing the best solution remains in the population [20], was also implemented.

Next, the crossover operation selects a pair of chromosomes and chooses a random point in the first chromosome. For the sections of both chromosomes from that point to the end of each chromosome, crossover exchanges machine assignments between corresponding tasks. Every chromosome is considered for crossover with a probability of 60%.

After crossover, the mutation operation is performed. Mutation randomly selects a task within the chromosome, and randomly reassigns it to a new machine. Both of these random operations select values from a uniform distribution. Every chromosome is considered for mutation with a probability of 40%.

Finally, the chromosomes from this modified population are evaluated again. This completes one iteration of the GA. The GA stops when any one of three conditions are met: (a) 1000 total iterations, (b) no change in the elite chromosome for 150 iterations, or (c) all chromosomes converge. If no stopping criteria is met, the loop repeats, beginning with the selection of a new population. The stopping criteria that usually occurred in testing was no change in the elite chromosome in 150 iterations.

SA: Simulated Annealing (SA) is an iterative technique that considers only one possible solution (mapping) for each meta-task at a time. This solution uses the same representation for a solution as the chromosome for the GA.

SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space (e.g., [6, 17, 21]). This probability is based on a system temperature that decreases for each iteration. As the system temperature "cools," it is more difficult for currently poorer solutions to be accepted. The initial system temperature is the makespan of the initial mapping.

The specific SA procedure implemented here is as follows. The initial mapping is generated from a uniform random distribution. The mapping is mutated in the same manner as the GA, and the new makespan is evaluated. The decision algorithm for accepting or rejecting the new mapping is based on [6]. If the new makespan is better, the new mapping replaces the old one. If the new makespan is worse (larger), a uniform random number $z \in [0, 1)$ is selected. Then, z is compared with y, where

$$y = \frac{1}{1 + e^{\left(\frac{\text{old makespan}-\text{new makespan}}{\text{temperature}}\right)}}. \quad (1)$$

If $z > y$ the new (poorer) mapping is accepted, otherwise it is rejected, and the old mapping is kept.

Notice that for solutions with similar makespans (or if the system temperature is very large), $y \to 0.5$, and poorer solutions are more easily accepted. In contrast, for solutions with very different makespans (or if the system temperature is very small), $y \to 1$, and poorer solutions will usually be rejected.

After each mutation, the system temperature is decreased by 10%. This defines one iteration of SA. The heuristic stops when there is no change in the makespan for 150 iterations or the system temperature reaches zero. Most tests ended with no change in the makespan

for 150 iterations.

GSA: The Genetic Simulated Annealing (GSA) heuristic is a combination of the GA and SA techniques [4, 23]. In general, GSA follows procedures similar to the GA outlined above. GSA operates on a population of 200 chromosomes, uses a Min-min seed in four out of eight initial populations, and performs similar mutation and crossover operations. However, for the selection process, GSA uses the SA cooling schedule and system temperature, and a simplified SA decision process for accepting or rejecting a new chromosomes. GSA also used elitism to guarantee that the best solution always remained in the population.

The initial system temperature for the GSA selection process was set to the average makespan of the initial population, and decreased 10% for each iteration. When a new (post-mutation, post-crossover, or both) chromosome is compared with the corresponding original chromosome, if the new makespan is less than the old makespan plus the system temperature, then the new chromosome is accepted. That is, if

$$\text{new makespan} < (\text{old makespan} + \text{temperature}) \quad (2)$$

is true, the new chromosome becomes part of the population. Otherwise, the original chromosome survives to the next iteration. Therefore, as the system temperature decreases, it is again more difficult for poorer solutions (i.e., longer makespans) to be accepted. The two stopping criteria used were either (a) no change in the elite chromosome in 150 iterations or (b) 1000 total iterations. Again, the most common stopping criteria was no change in the elite chromosome in 150 iterations.

Tabu: Tabu search is a solution space search that keeps track of the regions of the solution space which have already been searched so as not to repeat a search near these areas [7, 12]. A solution (mapping) uses the same representation as a chromosome in the GA approach.

The implementation of Tabu search used here begins with a random mapping, generated from a uniform distribution. Starting with the first task in the mapping, task $i = 0$, each possible pair of tasks is formed, (i, j) for $0 \leq i < t - 1$ and $i < j < t$. As each pair of tasks is formed, they exchange machine assignments. This constitutes a short hop. The intuitive purpose of a short hop is to find the nearest local minimum solution within the solution space. After each exchange, the new makespan is evaluated. If the new makespan is an improvement, the new mapping is accepted (a successful short hop), and the pair generation-and-exchange sequence starts over from the beginning ($i = 0$) of the new mapping. Otherwise, the pair generation-and-exchange sequence continues from its previous state, (i, j). New short hops are generated until 1200 successful short hops have been made or all combinations of task pairs have been exhausted with no further improvement.

At this point, the final mapping from the local solution space search is added to the tabu list. The tabu list is a method of keeping track of the regions of the solution space that have already been searched. Next, a new random mapping is generated, and it must differ from each mapping in the tabu list by at least half of the machine assignments (a successful long hop). The intuitive purpose of a long hop is to move to a new region of the solution space that has not already been searched. The final stopping criterion for the heuristic is a total of 1200 successful hops (short and long combined). Then, the best mapping from the tabu list is the final answer.

A*: The final heuristic in the comparison study is known as the A* heuristic. A* has been applied to many other task allocation problems (e.g., [5, 16, 21, 22]). The technique used here is similar to [5].

A* is a tree search beginning at a root node that is usually a null solution. As the tree grows, intermediate nodes represent partial solutions (a subset of tasks are assigned to machines), and leaf nodes represent final solutions (all tasks are assigned to machines). The partial solution of a child node has one more task mapped than the parent node. Call this additional task a. Each parent node generates m children, one for each possible mapping of a. After a parent node has done this, the parent node is removed and replaced in the tree by the m children. Based on experimentation and a desire to keep execution time of the heuristic tractable, the maximum number of nodes in the tree at any one time is limited in this study to $n_{max} = 1024$.

Each node, n, has a cost function, $f(n)$, associated with it. The cost function is an estimated lower-bound on the makespan of the best solution that includes the partial solution represented by node n. Let $g(n)$ represent the makespan of the task/machine assignments in the partial solution of node n, i.e., $g(n)$ is the maximum of the machine availability times ($mat(j)$) based on the set of tasks that have been mapped to machines in node n's partial solution. Let $h(n)$ be a lower-bound estimate on the difference between the makespan of node n's partial solution and the makespan for the best complete solution that includes node n's partial solution. Then, the cost function for node n is computed as

$$f(n) = g(n) + h(n). \quad (3)$$

Therefore, $f(n)$ represents the makespan of the partial solution of node n plus a lower-bound estimate of the

time to execute the rest of the (unmapped) tasks in the meta-task.

The function $h(n)$ is defined in terms of two functions, $h_1(n)$ and $h_2(n)$, which are two different approaches to deriving a lower-bound estimate. Recall that $M = \{m_i : m_i = \min_{0 \le j < m}(ct(i,j)),$ for each $i \in U\}$. For node n let $mmct(n)$ be the overall maximum element of M over all $i \in U$ (i.e., "the maximum minimum completion time"). Intuitively, $mmct(n)$ represents the best possible meta-task makespan by making the typically unrealistic assumption that each task in U can be assigned to the machine indicated in M without conflict. Thus, based on [5], $h_1(n)$ is defined as

$$h_1(n) = \max(0,\ (mmct(n) - g(n))). \qquad (4)$$

Next, let $sdma(n)$ be the sum of the differences between $g(n)$ and each machine availability time over all machines after executing all of the tasks in the partial solution represented by node n:

$$sdma(n) = \sum_{j=0}^{m-1} (g(n) - mat(j)). \qquad (5)$$

Intuitively, $sdma(n)$ represents the amount of machine availability time remaining that can be scheduled without increasing the final makespan. Let $smet(n)$ be defined as the sum of the minimum expected execution times (i.e., ETC values) for all tasks in U:

$$smet(n) = \sum_{i \in U} (\min_{0 \le j < m} (ETC(i,j)) \qquad (6)$$

This gives an estimate of the amount of remaining work to do, which could increase the final makespan. The function h_2 is then defined as

$$h_2(n) = \max(0,\ (smet(n) - sdma(n))/m), \qquad (7)$$

where $(smet(n) - sdma(n))/m$ represents an estimate of the minimum increase in the meta-task makespan if the tasks in U could be "ideally" (but, in general, unrealistically) distributed among the machines. Using these definitions,

$$h(n) = \max(h_1(n), h_2(n)), \qquad (8)$$

representing a lower-bound estimate on the time to execute the tasks in U.

Thus, beginning with the root, the node with the minimum $f(n)$ is replaced by its m children, until n_{max} nodes are created. From that point on, any time a node is added, the tree is pruned by deleting the node with the largest $f(n)$. This process continues until a leaf node (representing a complete mapping) is reached. Note that if the tree is not pruned, this method is equivalent to an exhaustive search.

These eleven heuristics were all implemented under the common simulation model described in Section 2. The results from experiments using these implementations are described in the next section. Suggestions for alternative heuristic implementations are given in Section 5.

4. Experimental Results

An interactive software application has been developed that allows simulation, testing, and demonstration of the heuristics examined in Section 3 applied to the meta-tasks defined by the ETC matrices described in Section 2. The software allows a user to specify t and m, to select which ETC matrices to use, and to choose which heuristics to execute. It then generates the specified ETC matrices, executes the desired heuristics, and displays the results, similar to Figures 2 through 13. The results discussed in this section were generated using portions of this software.

When comparing mapping heuristics, the execution time of the heuristics themselves is an important consideration. For the heuristics listed, the execution times varied greatly. The experimental results discussed below were obtained on a Pentium II 400 MHz processor with 1GB of RAM. Each of the simpler heuristics (OLB, UDA, Fast Greedy, and Greedy) executed in a few seconds for one ETC matrix with $t = 512$ and $m = 16$. For the same sized ETC matrix, SA and Tabu, both of which manipulate a single solution during an iteration, averaged less than 30 seconds. GA and GSA required approximately 60 seconds per matrix because they manipulate entire populations, and A* required about 20 minutes per matrix.

The resulting meta-task execution times (makespans) from the simulations for every case of consistency, task heterogeneity, and machine heterogeneity are shown in Figures 2 through 13. All experimental results represent the execution time of a meta-task (defined by a particular ETC matrix) based on the mapping found by the heuristic specified, averaged over 100 different ETC matrices of the same type (i.e., 100 mappings). For each heuristic, the range bars show the minimum and maximum meta-task execution times over the 100 mappings (100 ETC matrices) used to compute the average meta-task execution time. Tables 1 through 4 show sample subsections from the four types of inconsistent ETC matrices considered. Semi-consistent and consistent matrices of the same types could be generated from these matrices as described in Section 2. For the

results described here, however, entirely new matrices were generated for each case.

For the four consistent cases, Figures 2 through 5, the UDA algorithm had the worst execution times by an order of magnitude. This is easy to explain. For the consistent cases, all tasks will have the lowest execution time on one machine, and all tasks will be mapped to this particular machine. This corresponds to results found in [1]. Because of this poor performance, the UDA results were not included in Figures 2 through 5. OLB, Max-min, and SA had the next poorest results. GA performed the best for the consistent cases. This was due in part to the good performance of the Min-min heuristic. The best GA solution always came from one of the populations that had been seeded with the Min-min solution. As is apparent in the figures, Min-min performed very well on its own, giving the second best results. The mutation, crossover, and selection operations of the GA were always able to improve on this solution, however. GSA, which also used a Min-min seed, did not always improve upon the Min-min solution. Because of the probabilistic procedure used during selection, GSA would sometimes accept poorer intermediate solutions. These poorer intermediate solutions never led to better final solutions, thus GSA gave the third best results. The performance of A* was hindered because the estimates made by $h_1(n)$ and $h_2(n)$ are not as accurate for consistent cases as they are for inconsistent and semi-consistent cases. For consistent cases, $h_1(n)$ underestimates the competition for machines and $h_2(n)$ underestimates the "workload" distributed to each machine.

These results suggest that if the best overall solution is desired, the GA should be employed. However, the improvement of the GA solution over the Min-min solution was never more than 10%. Therefore, the Min-min hueristic may be more appropriate in certain situations, given the difference in execution times of the two heuristics.

For the four inconsistent test cases in Figures 6 through 9, UDA performs much better and the performance of OLB degrades. Because there is no pattern to the consistency, OLB will assign more tasks to poor or even worst-case machines, resulting in poorer schedules. In contrast, UDA improves because the "best" machines are distributed across the set of machines, thus task assignments will be more evenly distributed among the set of machines avoiding load imbalance. Similarly, Fast Greedy and Min-min performed very well, and slightly outperformed UDA, because the machines providing the best task completion times are more evenly distributed among the set of machines. Min-min was also better than Max-min for all of the inconsistent cases. The advantages Min-min gains by mapping "best case" tasks first outweighs the savings in penalties Max-min has by mapping "worst case" tasks first.

Tabu gave the second poorest results for the inconsistent cases, at least 16% poorer than the other heuristics. Inconsistent matrices generated more successful short hops than the associated consistent matrices. Therefore, fewer long hops were generated and less of the solution space was searched, resulting in poorer solutions. The increased number of successful short hops for inconsistent matrices can be explained as follows. The pairwise comparison procedure used by the short hop procedure will assign machines with better performance first, early in the search procedure. For the consistent cases, these machines will always be from the same set of machines. For inconsistent cases, these machines could be any machine. Thus, for consistent cases, the search becomes somewhat ordered, and the successful short hops get exhausted faster. For inconsistent cases, the lack of order means there are more successful short hops, resulting in fewer long hops.

GA and A* had the best average makespans, and were usually within a small constant factor of each other. The random approach employed by these methods was useful and helped overcome the difficulty of locating good mappings within inconsistent matrices. GA again benefited from having the Min-min initial mapping. A* did well because if the tasks get more evenly distributed among the machines, this more closely matches the lower-bound estimates of $h_1(n)$ and $h_2(n)$.

Finally, consider the semi-consistent cases in Figures 10 through 13. For semi-consistent cases with high machine heterogeneity, the UDA heuristic again gave the worst results. Intuitively, UDA is suffering from the same problem as in the consistent cases: half of all tasks are getting assigned to the same machine. OLB does poorly for high machine heterogeneity cases because worst case matchings will have higher execution times for high machine heterogeneity. For low machine heterogeneity, the worst case matchings have a much lower penalty. The best heuristics for the semi-consistent cases were Min-min and GA. This is not surprising because these were two of the best heuristics from the consistent and inconsistent tests, and semi-consistent matrices are a combination of consistent and inconsistent matrices. Min-min was able to do well because it searched the entire row for each task and assigned a high percentage of tasks to their first choice. GA was robust enough to handle the consistent components of the matrices, and did well for the same reasons mentioned for inconsistent matrices.

5. Alternative Implementations

The experimental results in Section 4 show the performance of each heuristic under the assumptions presented. For several heuristics, specific control parameter values and control functions had to be selected. In most cases, control parameter values and control functions were based on the references cited or experiments conducted. However, for these heuristics, different, valid implementations are possible using different control parameters and control functions.

GA, SA, GSA: Several parameter values could be varied among these techniques, including (where appropriate) population size, crossover probability, mutation probability, stopping criteria, number of runs with different initial populations per result, and the system temperature. The specific procedures used for the following actions could also be modified (where appropriate) including initial population "seed" generation, mutation, crossover, selection, elitism, and the accept/reject new mapping procedure.

Tabu: The short hop method implemented was a "first descent" (take the first improvement possible) method. "Steepest descent" methods (where several short hops are considered simultaneously, and the one with the most improvement is selected) are also used in practice [7]. Other techniques that could be varied are the long hop method, the order of the short hop pair generation-and-exchange sequence, and the stopping condition. Two possible alternative stopping criteria are when the tabu list reaches a specified number of entries, or when there is no change in the best solution in a specified number of hops.

A*: Several variations of the A* method that was employed here could be implemented. Different functions could be used to estimate the lower-bound $h(n)$. The maximum size of the search tree could be varied, and several other techniques exist for tree pruning (e.g., [21]).

In summary, for the GA, SA, GSA, Tabu, and A* heuristics there are a great number of possible valid implementations. An attempt was made to use a reasonable implementation of each heuristic for this study. Future work could examine other implementations.

6. Conclusions

The goal of this study was to provide a basis for comparison and insights into circumstances where one technique will out perform another for eleven different heuristics. The characteristics of the *ETC* matrices used as input for the heuristics and the methods used to generate them were specified. The implementation of a collection of eleven heuristics from the literature was described. The results of the mapping heuristics were discussed, revealing the best heuristics to use in certain scenarios. For the situations, implementations, and parameter values used here, GA was the best heuristic for most cases, followed closely by Min-min, with A* also doing well for inconsistent matrices.

A software tool was developed that allows others to compare these heuristics for many different types of *ETC* matrices. These heuristics could also be the basis of a mapping toolkit. If this toolkit were given an *ETC* matrix representing an actual meta-task and an actual HC environment, the toolkit could analyze the *ETC* matrix, and utilize the best mapping heuristic for that scenario. Depending on the overall situation, the execution time of the mapping heuristic itself may impact this decision. For example, if the best mapping available in less than one minute was desired and if the characteristics of a given *ETC* matrix most closely matched a consistent matrix, Min-min would be used; if more time was available for finding the best mapping, GA and A* should be considered.

The comparisons of the eleven heuristics and twelve situations provided in this study can be used by researchers as a starting point when choosing heuristics to apply in different scenarios. They can also be used by researchers for selecting heuristics to compare new, developing techniques against.

Acknowledgments – The authors thank Shoukat Ali and Taylor Kidd for their comments.

References

[1] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 79–87.

[2] R. Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Sept. 1997 (D. Hensgen, advisor.)

[3] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems," *IEEE Workshop on Advances in Parallel and Distributed Systems*, Oct. 1998, pp. 330–335

(included in the Proceedings of the *7th IEEE Symposium on Reliable Distributed Systems*, 1998).

[4] H. Chen, N. S. Flann, and D. W. Watson, "Parallel genetic simulated annealing: A massively parallel SIMD approach," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 9, No. 2, Feb. 1998, pp. 126–136.

[5] K. Chow and B. Liu, "On mapping signal processing algorithms to a heterogeneous multiprocessor system," *1991 International Conference on Acoustics, Speech, and Signal Processing - ICASSP 91*, Vol. 3, May 1991, pp. 1585–1588.

[6] M. Coli and P. Palazzari, "Real time pipelined system design through simulated annealing," *Journal of Systems Architecture*, Vol. 42, No. 6-7, Dec. 1996, pp. 465–475.

[7] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro, "Improving search by incorporating evolution principles in parallel tabu search," *1994 IEEE Conference on Evolutionary Computation*, Vol. 2, 1994, pp. 823–828.

[8] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, Nov. 1989, pp. 1427–1436.

[9] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 184–199.

[10] R. F. Freund and H. J. Siegel, "Heterogeneous processing," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 13–17.

[11] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 78–86.

[12] F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, MA, June 1997.

[13] D. A. Hensgen, T. Kidd, M. C. Schnaidt, D. St. John, H. J. Siegel, T. D. Braun, M. Maheswaran, S. Ali, J-K. Kim, C. Irvine, T. Levin, R. Wright, R. F. Freund, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, "An overview of MSHN: A Management System for Heterogeneous Networks," *8th IEEE Workshop on Heterogeneous Computing Systems (HCW '99)*, Apr. 1999, to appear.

[14] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 18–27.

[15] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, Vol. 24, No. 2, Apr. 1977, pp. 280–289.

[16] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, Vol. 6, No. 3, July–Sept. 1998, pp. 42–51.

[17] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671–680.

[18] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," *Encyclopedia of Electrical and Electronics Engineering*, J. Webster, ed., John Wiley & Sons, New York, NY, to appear 1999.

[19] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[20] G. Rudolph, "Convergence analysis of canonical genetic algorithms," *IEEE Transactions on Neural Networks*, Vol. 5, No. 1, Jan. 1994, pp. 96–101.

[21] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[22] C.-C. Shen and W.-H. Tsai, "A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion," *IEEE Transactions on Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 197–203.

[23] P. Shroff, D. Watson, N. Flann, and R. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments," *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, April 1996, pp. 98–104.

[24] H. J. Siegel, H. G. Dietz, and J. K. Antonio, "Software support for heterogeneous computing," in *The Computer Science and Engineering Handbook*, A. B. Tucker, Jr., ed., CRC Press, Boca Raton, FL, 1997, pp. 1886–1909.

[25] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, Apr. 1996, pp. 86–97.

[26] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *IEEE Computer*, Vol. 27, No. 6, June 1994, pp. 17–26.

[27] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, Vol. 47, No. 1, Nov. 1997, pp. 1–15.

Biographies

Tracy D. Braun is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received a Bachelor of Science in Electrical Engineering with Honors and High Distinction from the University of Iowa in 1995. In 1997, he received an MSEE from the School of Electrical and Computer Engineering at Purdue University. He received a Benjamin Meisner Fellowship from Purdue University for the 1995-1996 academic year. He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu honorary society. He is an active member of the Beta Chapter of Eta Kappa Nu at Purdue University, and has held several offices during his studies at Purdue, including chapter President. He has also been employed at Norand Data Systems and Silicon Graphics Inc./Cray Research. His research interests include parallel algorithms, heterogeneous computing, computer security, and software design.

H. J. Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE and a Fellow of the ACM. He received BS degrees in both Electrical Engineering and Management from MIT, and the MA, MSE, and PhD degrees from the Department of Electrical Engineering and Computer Science at Princeton University. Prof. Siegel has coauthored over 250 technical papers, has coedited seven volumes, and wrote the book *Interconnection Networks for Large-Scale Parallel Processing*. He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing*, and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He was Program Chair/Co-Chair of three conferences, General Chair/Co-Chair of four conferences, and Chair/Co-Chair of four workshops. He is an international keynote speaker and tutorial lecturer, and a consultant for government and industry.

Noah Beck is a Research Assistant and MSEE student at Purdue University in the School of Electrical and Computer Engineering. He received a Bachelor of Science in Computer Engineering from Purdue University in 1997, and is an active member of the Beta chapter of the Eta Kappa Nu honorary society. He has also been employed at Intel Corporation, and his research interests include microprocessor architecture, parallel computing, and heterogeneous computing.

Ladislau L. Bölöni is a PhD student and Research Assistant in the Computer Sciences Department at Purdue University. He received a Diploma Engineer degree in Computer Engineering with Honors from the Technical University of Cluj-Napoca, Romania in 1993. He received a fellowship from the Hungarian Academy of Sciences for the 1994-95 academic year. He is a member of ACM and the Upsilon Pi Epsilon honorary society. His research interests include distributed object systems, autonomous agents and parallel computing.

Muthucumaru Maheswaran is an Assistant Professor in the Department of Computer Science at the University of Manitoba, Canada. In 1990, he received a BSc degree in Electrical and Electronic Engineering from the University of Peradeniya, Sri Lanka. He received an MSEE degree in 1994 and a PhD degree in 1998, both from the School of Electrical and Computer Engineering at Purdue University. He held a Fulbright scholarship during his tenure as an MSEE student at Purdue University. His research interests include computer architecture, distributed computing, heterogeneous computing, Internet and World Wide Web systems, metacomputing, mobile programs, network computing, parallel computing, resource management systems for metacomputing, and scientific computing. He has authored or coauthored 15 technical papers in these and related areas. He is a member of the Eta Kappa Nu honorary society.

Albert I. Reuther is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received his Bachelor of Science in Computer and Electrical Engineering with Highest Distinction in 1994 and received a Masters of Science in Electrical Engineering in 1996, both at Purdue. He was a Purdue Andrews Fellowship recipient in the 1994-95 and 1995-96 academic years. He is a member of IEEE, IEEE Computer Society, ACM, and Eta Kappa Nu honorary society and has been employed by General Motors and Hewlett-Packard. His research interests include multimedia systems, heterogeneous computing, parallel processing, and educational multimedia.

James P. Robertson currently works for Motorola's PowerPC System Performance and Modeling group. He received a Bachelor of Science in Computer Engineering with Honors from the school of Electrical and Computer Engineering at Purdue University in 1996. As an undergraduate student he received an NSF undergraduate research scholarship. In 1998 he received an MSEE from Purdue University. He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu honorary society. While attending Purdue

University he was an active member of the Beta Chapter of Eta Kappa Nu, having held several offices including chapter Treasurer.

Mitchell D. Theys is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received a Bachelor of Science in Computer and Electrical Engineering in 1993 with Highest Distinction, and a Master of Science in Electrical Engineering in 1996, both from Purdue. He received support from a Benjamin Meisner Fellowship from Purdue University, an Intel Graduate Fellowship, and an AFCEA Graduate Fellowship. He is a member of the Eta Kappa Nu honorary society, IEEE, and IEEE Computer Society. He was elected President of the Beta Chapter of Eta Kappa Nu at Purdue University and has held several various offices during his stay at Purdue. He has held positions with Compaq Computer Corporation, S&C Electic Company, and Lawrence Livermore National Laboratory. His research interests include design of single chip parallel machines, heterogeneous computing, parallel processing, and software/hardware design.

Bin Yao is a PhD student and Research Assistant in the School of Electrical and Computer Engineering at Purdue University. He received Bachelor of Science in Electrical Engineering from Beijing University in 1996. He received Andrews Fellowship from Purdue University for the academic years 1996-1998. He is a student member of the IEEE. His research interests include distributed algorithms, fault tolerant computing, and heterogeneous computing.

Debra Hensgen is an Associate Professor in the Computer Science Department at The Naval Postgraduate School. She received her PhD in the area of Distributed Operating Systems from the University of Kentucky. She is currently a Principal Investigator of the DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) and a co-investigator of the DARPA-sponsored Server and Active Agent Management (SAAM) Next Generation Internet project. Her areas of interest include active modeling in resource management systems, network re-routing to preserve quality of service guarantees, visualization tools for performance debugging of parallel and distributed systems, and methods for aggregating sensor information. She has published numerous papers concerning her contributions to the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems.

Richard F. Freund is a founder and CEO of NOEMIX, a San Diego based startup to commercialize distributed computing technology. Dr. Freund is also one of the early pioneers in the field of distributed computing, in which he has written or co-authored a number of papers. In addition he is a founder of the Heterogeneous Computing Workshop, held each year in conjunction with IPPS/SPDP. Freund won a Meritorious Civilian Service Award during his former career as a government scientist.

```
initial population generation;
evaluation;
while (stopping criteria not met) {
        selection;
        crossover;
        mutation;
        evaluation;
}
```

Figure 1. General procedure for a Genetic Algorithm, based on [26].

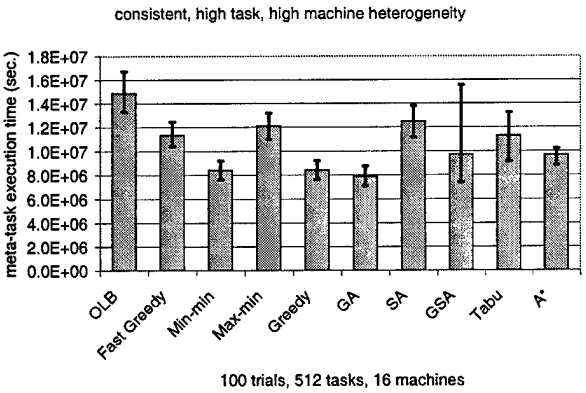


Figure 2. Consistent, high task, high machine heterogeneity.

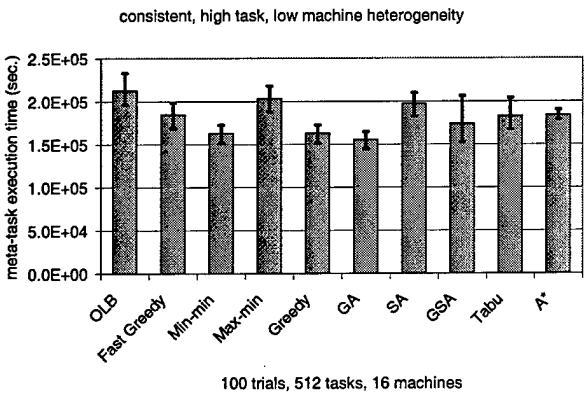


Figure 3. Consistent, high task, low machine heterogeneity.

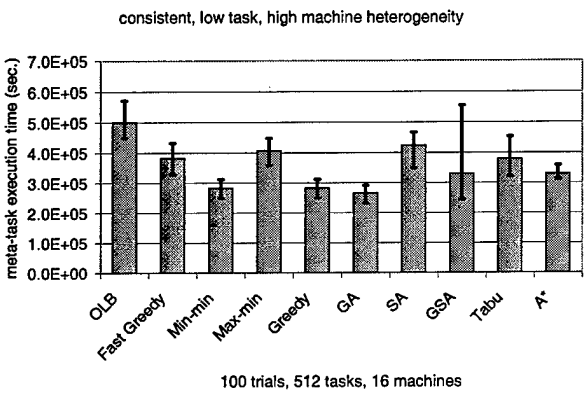


Figure 4. Consistent, low task, high machine heterogeneity.

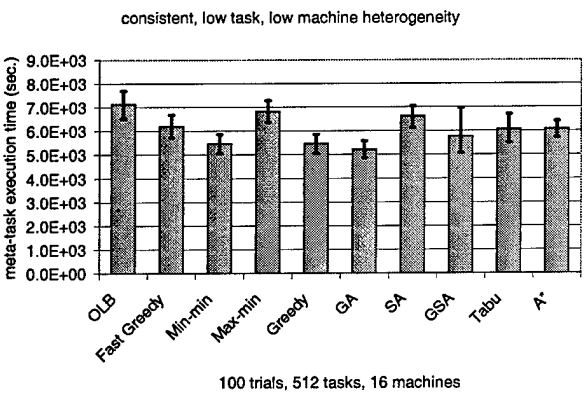


Figure 5. Consistent, low task, low machine heterogeneity.

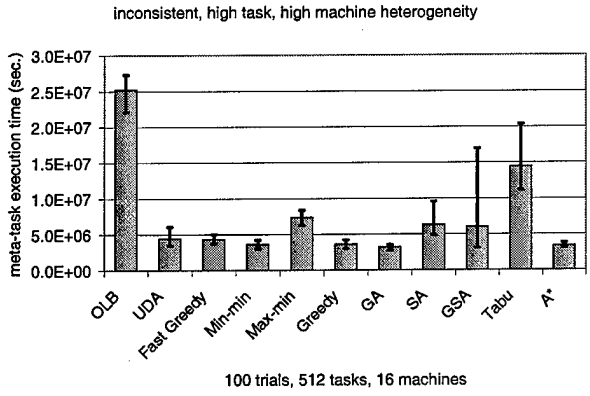


Figure 6. Inconsistent, high task, high machine heterogeneity.

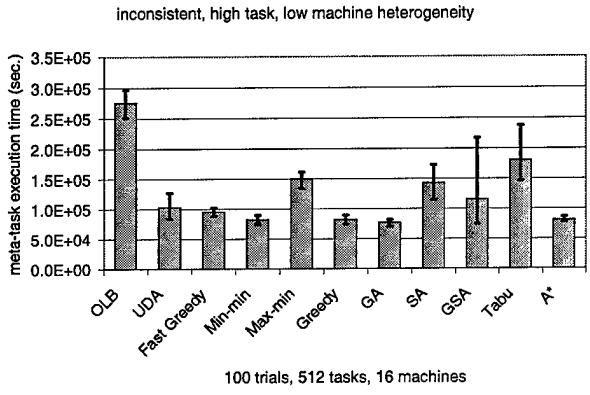


Figure 7. Inconsistent, high task, low machine heterogeneity.

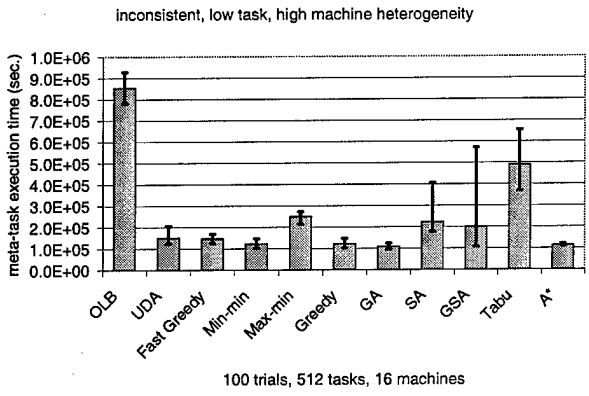


Figure 8. Inconsistent, low task, high machine heterogeneity.

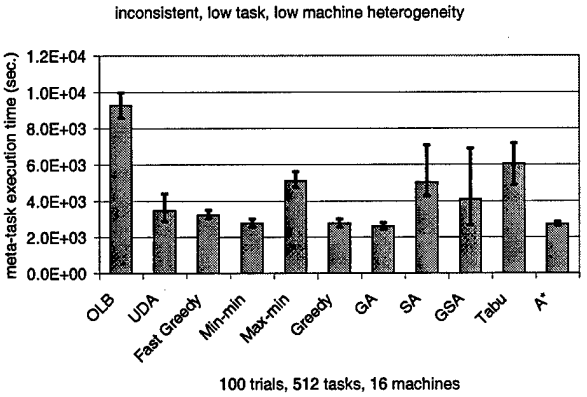


Figure 9. Inconsistent, low task, low machine heterogeneity.

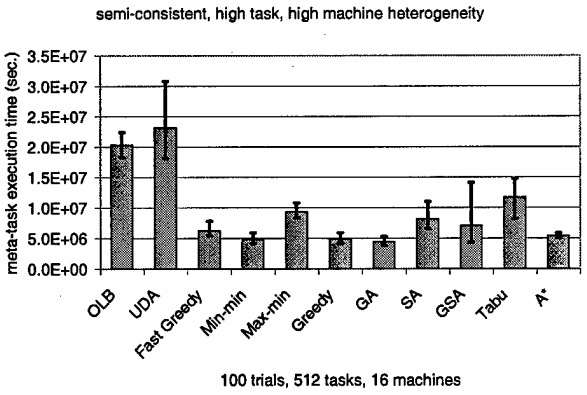


Figure 10. Semi-consistent, high task, high machine heterogeneity.

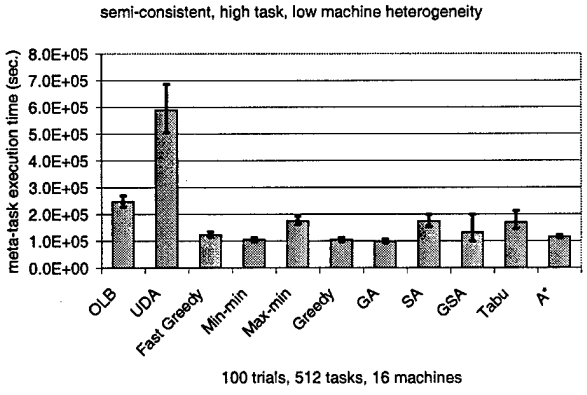


Figure 11. Semi-consistent, high task, low machine heterogeneity.

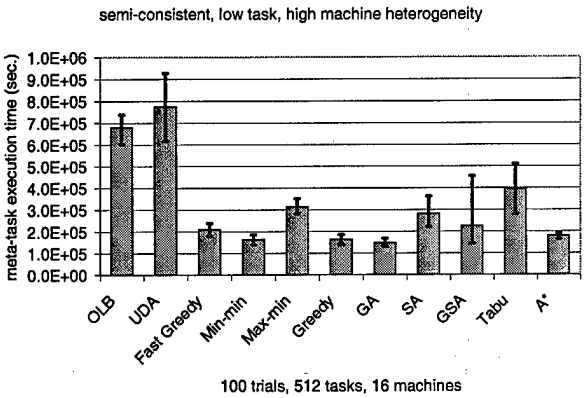


Figure 12. Semi-consistent, low task, high machine heterogeneity.

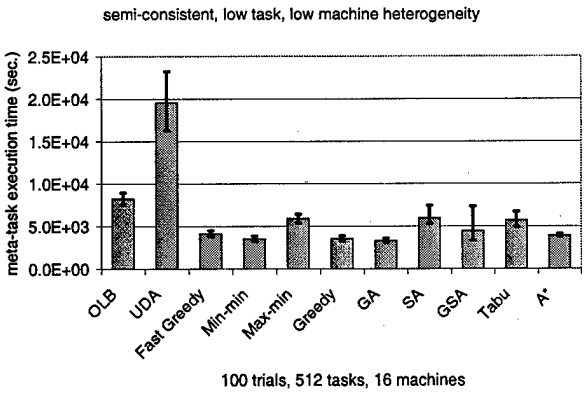


Figure 13. Semi-consistent, low task, low machine heterogeneity.

	machines							
t	436,735.9	815,309.1	891,469.0	1,722,197.6	1,340,988.1	740,028.0	1,749,673.7	251,140.1
a	950,470.7	933,830.1	2,156,144.2	2,202,018.0	2,286,210.0	2,779,669.0	220,536.3	1,769,184.5
s	453,126.6	479,091.9	150,324.5	386,338.1	401,682.9	218,826.0	242,699.6	11,392.2
k	1,289,078.2	1,400,308.1	2,378,363.0	2,458,087.0	351,387.4	925,070.1	2,097,914.2	1,206,158.2
s	646,129.6	576,144.9	1,475,908.2	424,448.8	576,238.7	223,453.8	256,804.5	88,737.9
	1,061,682.3	43,439.8	1,355,855.5	1,736,937.1	1,624,942.6	2,070,705.1	1,977,650.2	1,066,470.8
	10,783.8	7,453.0	3,454.4	23,720.8	29,817.3	1,143.7	44,249.2	5,039.5
	1,940,704.5	1,682,338.5	1,978,545.6	788,342.1	1,192,052.5	1,022,914.1	701,336.3	1,052,728.3

Table 1. Sample 8×8 excerpt from ETC with inconsistent, high task, high machine heterogeneity.

	machines							
t	21,612.6	13,909.7	6,904.1	3,621.5	3,289.5	8,752.0	5,053.7	14,515.3
a	578.4	681.1	647.9	477.1	811.9	619.5	490.9	828.7
s	122.8	236.9	61.3	143.6	56.0	313.4	283.5	241.9
k	1,785.7	1,528.1	6,998.8	4,265.3	3,174.6	3,438.0	7,168.4	2,059.3
s	510.8	472.0	358.5	461.4	1,898.7	1,535.4	1,810.2	906.6
	22,916.7	18,510.0	11,932.7	6,088.3	9,239.7	15,036.4	18,107.7	12,262.6
	5,985.3	2,006.5	1,546.4	6,444.6	2,640.0	7,389.3	5,924.9	1,867.2
	16,192.4	3,088.9	16,532.5	13,160.6	10,574.2	7,136.3	15,353.4	2,150.6

Table 2. Sample 8×8 excerpt from ETC with inconsistent, high task, low machine heterogeneity.

	machines							
t	16,603.2	71,369.1	39,849.0	44,566.1	55,124.3	9,077.3	87,594.5	31,530.5
a	738.3	2,375.0	5,606.2	804.9	1,535.8	4,772.3	994.2	1,833.9
s	1,513.8	45.1	1,027.3	2,962.1	2,748.2	2,406.3	19.4	969.9
k	2,219.9	5,989.2	2,747.0	88.2	2,055.1	665.0	356.3	2,404.9
s	12,654.7	10,483.7	10,601.5	6,804.6	134.3	10,532.8	12,341.5	5,046.3
	4,226.0	48,152.2	11,279.3	35,471.1	30,723.4	24,234.0	6,366.9	22,926.9
	20,668.5	28,875.9	29,610.1	7,363.3	24,488.0	31,077.3	8,705.0	11,849.4
	52,953.2	14,608.1	58,137.2	16,685.5	36,571.3	35,888.8	38,147.0	15,167.5

Table 3. Sample 8×8 excerpt from ETC with inconsistent, low task, high machine heterogeneity.

	machines							
t	512.9	268.0	924.9	494.4	611.2	606.9	921.6	209.6
a	8.5	16.8	23.4	19.2	27.9	22.7	19.6	8.3
s	228.8	238.5	107.2	180.0	334.6	88.2	192.8	125.7
k	345.1	642.4	136.8	206.2	559.5	349.5	640.2	664.2
s	117.3	235.9	149.9	71.5	136.6	363.6	182.8	359.5
	240.7	412.0	259.1	319.8	237.5	338.3	178.5	537.7
	462.8	93.3	574.9	449.4	421.8	559.6	487.7	298.7
	119.5	36.7	224.2	194.2	176.5	156.8	182.7	192.0

Table 4. Sample 8×8 excerpt from ETC with inconsistent, low task, low machine heterogeneity.

Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems

Muthucumaru Maheswaran[†], Shoukat Ali[‡], Howard Jay Siegel[‡], Debra Hensgen[◇], and Richard F. Freund[⋆]

[†]Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2, Canada
Email: maheswar@cs.umanitoba.ca

[‡] School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285 USA
Email: {alis,hj}@ecn.purdue.edu

[◇]Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940 USA
Email: hensgen@cs.nps.navy.mil

[⋆]NOEMIX Inc.
1425 Russ Blvd., Ste. T-110
San Diego, CA 92101 USA
Email: rffreund@noemix.com

Abstract

Dynamic mapping (matching and scheduling) heuristics for a class of independent tasks using heterogeneous distributed computing systems are studied. Two types of mapping heuristics are considered: on-line and batch mode heuristics. Three new heuristics, one for batch and two for on-line, are introduced as part of this research. Simulation studies are performed to compare these heuristics with some existing ones. In total, five on-line heuristics and three batch heuristics are examined. The on-line heuristics consider, to varying degrees and in different ways, task affinity for different machines and machine ready times. The batch heuristics consider these factors, as well as aging of tasks waiting to execute. The simulation results reveal that the choice of mapping heuristic depends on parameters such as: (a) the structure of the heterogeneity among tasks and machines, (b) the optimization requirements, and (c) the arrival rate of the tasks.

1. Introduction

An emerging trend in computing is to use distributed heterogeneous computing (HC) systems constructed by networking various machines to execute a set of tasks [5, 14]. These HC systems have resource management systems (RMSs) to govern the execution of the tasks that arrive for service. This paper describes and compares eight heuristics that can be used in such an RMS for assigning tasks to machines.

In a general HC system, dynamic schemes are necessary to assign tasks to machines (matching), and to compute the execution order of the tasks assigned to each machine (scheduling) [3]. In the HC system considered here, the tasks are assumed to be independent, i.e., no communications between the tasks are needed. A dynamic scheme is needed because the arrival times of the tasks may be random and some machines in the suite may go off-line and new machines may come on-line. The dynamic mapping (matching and scheduling) heuristics investigated in this study are non-preemptive, and assume that the tasks have no deadlines or priorities associated with them.

The mapping heuristics can be grouped into two categories: on-line mode and batch-mode heuristics. In the on-line mode, a task is mapped onto a machine as soon as it arrives at the mapper. In the batch mode, tasks are not mapped onto the machines as they arrive; instead they are collected into a set that is examined for mapping at prescheduled times called mapping events. The independent set of tasks that is considered for mapping at the mapping events is called a meta-task. A meta-task can include newly arrived tasks (i.e., the ones arriving after the last mapping event) and the ones that were mapped in earlier mapping events but did not begin execution. While on-line mode heuristics consider a task for mapping only once, batch mode heuristics consider a task for mapping at each mapping event until the task begins execution.

The trade-offs between on-line and batch mode heuristics are studied experimentally. Mapping independent tasks onto an HC suite is a well-known NP-complete problem if throughput is the optimization criterion [9]. For the heuristics discussed in this paper, maximization of the throughput is the primary objective. This performance metric is the most common one in the production oriented environments. However, the performance of the heuristics is examined us-

This research was supported by the DARPA/ITO Quorum Program under the NPS subcontract numbers N62271-97-M-0900, N62271-98-M-0217, and N62271-98-M-0448. Some of the equipment used was donated by Intel.

0-7695-0107-9/99 $10.00 © 1999 IEEE

ing other metrics as well.

Three new heuristics, one for batch and two for on-line, are introduced as part of this research. Simulation studies are performed to compare these heuristics with some existing ones. In total, five on-line heuristics and three batch heuristics are examined. The on-line heuristics consider, to varying degrees and in different ways, task affinity for different machines and machine ready times. The batch heuristics consider these factors, as well as aging of tasks waiting to execute.

Section 2 describes some related work. In Section 3, the optimization criterion and another performance metric are defined. Section 4 discusses the mapping approaches studied here. The simulation procedure is given in Section 5. Section 6 presents the simulation results.

This research is part of a DARPA/ITO Quorum Program project called MSHN (Management System for Heterogeneous Networks) [8]. MSHN is a collaborative research effort that includes Naval Postgraduate School, NOEMIX, Purdue, and University of Southern California. It builds on SmartNet, an operational scheduling framework and system for managing resources in an HC environment developed at NRaD [6]. The technical objective of the MSHN project is to design, prototype, and refine a distributed resource management system that leverages the heterogeneity of resources and tasks to deliver the requested qualities of service. The heuristics developed here, or their derivatives, may be included in the Scheduling Advisor component of the MSHN prototype.

2. Related Work

In the literature, mapping tasks onto machines is often referred to as scheduling. Several researchers have worked on the dynamic mapping problem from areas including job shop scheduling and distributed computer systems (e.g., [10, 12, 18, 20]).

Some of the heuristics examined for batch-mode mapping in this paper are based on the static heuristics given in [9]. The heuristics presented in [9] are concerned with mapping independent tasks onto heterogeneous machines such that the completion time of the last finishing task is minimized. The problem is recognized as NP-complete and several heuristics are designed. Worst case performance bounds are obtained for the heuristics. The Min-min heuristic that is used here as a benchmark for batch mode mapping is based on the ideas presented in [9], and implemented in SmartNet [6].

In [10], a dynamic matching and scheduling scheme based on a distributed policy for mapping tasks onto HC systems is provided. A task can have several subtasks, and the subtasks can have data dependencies among them. In the scheme presented in [10], the subtasks in an application receive information about the subtasks in other applications only in terms of load estimates on the machines. Each application uses an algorithm that uses a weighting factor to determine the mapping for the subtasks. The weighting factor for a subtask is derived by considering the length of the critical path from the subtask to the end of the directed acyclic graph (DAG) that represents the application. If each application is an independent task with no subtasks, as is the case in this paper, then the scheme presented in [10] is not suitable, because the mapping criterion is designed to exploit information available in a DAG. Therefore, the scheme provided in [10] is not compared to the heuristics presented in this paper.

Two dynamic mapping approaches, one using a centralized policy and the other using a distributed policy, are developed in [12]. The centralized heuristic referred to therein as the global queue equalization algorithm is similar to the minimum completion time heuristic that is used as a benchmark in this paper and described in Section 4. The heuristic based on the distributed policy uses a method similar to the minimum completion time heuristic at each node. The mapper at a given node considers the local machine and the k highest communication bandwidth neighbors to map the tasks in the local queue. Therefore, the mapper based on the distributed strategy assigns a task to the best machine among the $k+1$ machines. The simulation results provided in [12] show that the centralized heuristic always performs better than the distributed heuristic. The heuristics in [12] are very similar to the minimum completion time heuristic used as a benchmark in this paper. Hence, they are not experimentally compared with the heuristics presented here.

In [18], a survey of dynamic scheduling heuristics for distributed computing systems is provided. Most of the heuristics featured in [18] perform load sharing to schedule the tasks on different machines, not considering any task-machine affinities while making the mapping decisions for HC systems. In contrast to [18], these affinities are considered to varying degrees in all but one of the heuristics examined in this paper.

A survey of dynamic scheduling heuristics for job-shop environments is provided in [20]. It classifies the dynamic scheduling algorithms into three approaches: conventional approach, knowledge-based approach, and distributed problem solving approach. The class of heuristics grouped under the conventional approach are similar to the minimum completion time heuristic considered in this paper, however, the problem domains considered in [20] and here differ. Furthermore, some of the heuristics featured in [20] use priorities and deadlines to determine the task scheduling order whereas priorities and deadlines are not considered here.

In distributed computer systems, load balancing schemes have been a popular strategy for mapping tasks onto the machines (e.g., [15, 18]). In [15], the performance characteristics of simple load balancing heuristics for HC distributed

systems are studied. The heuristics presented in [15] do not consider task execution times when making their decisions.

SmartNet [6] is an RMS for HC systems that employs various heuristics to map tasks to machines considering resource and task heterogeneity. In this paper, some appropriate selected SmartNet heuristics are included in the comparative study.

3. Performance Metrics

The expected execution time e_{ij} of task t_i on machine m_j is defined as the amount of time taken by m_j to execute t_i given m_j has no load when t_i is assigned. The expected completion time c_{ij} of task t_i on machine m_j is defined as the wall-clock time at which m_j completes t_i (after having finished any previously assigned tasks). Let m be the total number of the machines in the HC suite. Let K be the set containing the tasks that will be used in a given test set for evaluating heuristics in the study. Let the arrival time of the task t_i be a_i, and let the begin time of t_i be b_i. From the above definitions, $c_{ij} = b_i + e_{ij}$. Let c_i be c_{ij}, where machine j is assigned to execute task i. The makespan for the complete schedule is then defined as $max_{t_i \in K}(c_i)$ [17]. Makespan is a measure of the throughput of the HC system, and does not measure the quality of service imparted to an individual task.

Recall from Section 1, in on-line mode, the mapper assigns a task to a machine as soon as the task arrives at the mapper, and in batch mode a set of independent tasks that need to be mapped at a mapping event is called a meta-task. (In some systems, the term meta-task is defined in a way that allows inter-task dependencies.) In batch mode, for the i-th mapping event, the meta-task M_i is mapped at time τ_i, where $i \geq 0$. The initial meta-task, M_0, consists of all the tasks that arrived prior to time τ_0, i.e., $M_0 = \{t_j \mid a_j < \tau_0\}$. The meta-task, M_k, for $k > 0$, consists of tasks that arrived after the last mapping event and the tasks that had been mapped, but did not start executing, i.e., $M_k = \{t_j \mid \tau_{k-1} \leq a_j < \tau_k\} \cup \{t_j \mid a_j < \tau_{k-1}, b_j > \tau_k\}$. The waiting time for task t_j is defined as $b_j - a_j$. Let $\bar{c}_j$ be the completion time of task t_j if it is the only task that is executing on the system. The sharing penalty (ρ_j) for the task t_j is defined as $(c_j - \bar{c}_j)$. The average sharing penalty for the tasks in the set K is given by $[\sum_{t_j \in K} \rho_j] / \mid K \mid$. The average sharing penalty for a set of tasks mapped by a given heuristic is an indication of the heuristic's ability to minimize the effects of contention among different tasks in the set. It therefore indicates quality of service provided to an individual task, as gauged by the wait incurred by the task before it begins and the time to perform the actual computation. Other performance metrics are considered in [13].

4. Mapping Heuristics

4.1. Overview

In the on-line mode heuristics, each task is considered only once for matching and scheduling, i.e., the mapping is not changed once it is computed. When the arrival rate is low, machines may be ready to execute a task as soon as it arrives at the mapper. Therefore, it may be beneficial to use the mapper in the on-line mode so that a task need not wait until the next mapping event to begin its execution.

In batch mode, the mapper considers a meta-task for matching and scheduling at each mapping event. This enables the mapping heuristics to possibly make better decisions, because the heuristics have the resource requirement information for a whole meta-task, and know about the actual execution times of a larger number of tasks (as more tasks might complete while waiting for the mapping event). When the task arrival rate is high, there will be a sufficient number of tasks to keep the machines busy in between the mapping events, and while a mapping is being computed. It is, however, assumed in this study that the running time of the heuristic is negligibly small as compared to the average task execution time.

Both on-line and batch mode heuristics assume that estimates of expected task execution times on each machine in the HC suite are known. The assumption that these estimated expected times are known is commonly made when studying mapping heuristics for HC systems (e.g.,[7, 11, 19]). (Approaches for doing this estimation based on task profiling and analytical benchmarking are discussed in [14].) These estimates can be supplied before a task is submitted for execution, or at the time it is submitted. (The use of some of the heuristics studied here in a static environment is discussed in [4].)

The ready time of a machine is quantified by the earliest time that machine is going to be ready after completing the execution of the tasks that are currently assigned to it. It is assumed that each time a task t_i completes on a machine m_j a report is sent to the mapper. Because the heuristics presented here are dynamic, the expected machine ready times are based on a combination of actual task execution times and estimated expected task execution times. The experiments presented in Section 6 model this situation using simulated actual values for the execution times of the tasks that have already finished their execution. Also, all heuristics examined here operate in a centralized fashion on a dedicated suite of machines; i.e., the mapper controls the execution of all jobs on all machines in the suite. It is also assumed that the mapping heuristic is being run on a separate machine.

4.2. On-line mode mapping heuristics

The MCT (minimum completion time) heuristic assigns each task to the machine that results in that task's earliest completion time. This causes some tasks to be assigned to machines that do not have the minimum execution time for them. The MCT heuristic is a variant of the fast-greedy heuristic from SmartNet [6]. The MCT heuristic is used as a benchmark for the on-line mode, i.e., the performance of the other heuristics is compared against that of the MCT heuristic.

As a task arrives, all the machines in the HC suite are examined to determine the machine that gives the earliest completion time for the task. Therefore, it takes $O(m)$ time to map a given task.

The MET (minimum execution time) heuristic assigns each task to the machine that performs that task's computation in the least amount of execution time (this heuristic is also known as LBA (limited best assignment) [1] and UDA (user directed assignment) [6]). This heuristic, in contrast to MCT, does not consider machine ready times. This heuristic can cause a severe imbalance in load across the machines. The advantages of this method are that it gives each task to the machine that performs it in the least amount of execution time, and the heuristic is very simple. The heuristic needs $O(m)$ time to find the machine that has the minimum execution time for a task.

The SA (switching algorithm) heuristic is motivated by the following observations. The MET heuristic can potentially create load imbalance across machines by assigning many more tasks to some machines than to others, whereas the MCT heuristic tries to balance the load by assigning tasks for earliest completion time. If the tasks are arriving in a random mix, it is possible to use MET at the expense of load balance until a given threshold and then use MCT to smooth the load across the machines. The SA heuristic uses the MCT and MET heuristics in a cyclic fashion depending on the load distribution across the machines. The purpose is to have a heuristic with the desirable properties of both the MCT and the MET.

Let the maximum ready time over all machines in the suite be r_{max}, and the minimum ready time be r_{min}. Then, the load balance index across the machines is given by $\pi = r_{min}/r_{max}$. The parameter π can have any value in the interval $[0, 1]$. If π is 1.0, then the load is evenly balanced across the machines. If π is 0, then at least one machine has not yet been assigned a task. Two threshold values, π_l (low) and π_h (high), for the ratio π are chosen in $[0, 1]$ such that $\pi_l < \pi_h$. Initially, the value of π is set to 0.0. The SA heuristic begins mapping tasks using the MCT heuristic until the value of load balance index increases to at least π_h. After that point in time, the SA heuristic begins using the MET heuristic to perform task mapping. This causes the load balance index to decrease. When it reaches π_l, the SA heuristic switches back to using the MCT heuristic for mapping the tasks and the cycle continues.

As an example of functioning of the SA with lower and upper limits of 0.6 and 0.9, respectively, for $| K | = 1000$, the SA switched between the MET and the MCT two times, assigning 715 tasks using the MCT. For $| K | = 2000$, the SA switched five times, using the MCT to assign 1080 tasks. The percentage of tasks assigned using MCT gets progressively smaller for larger $| K |$. This is because an MET assignment in a highly loaded system will bring a smaller decrease in load balance index than when the same assignment is made in a lightly loaded system. Therefore many more MET assignments can be made in a highly loaded system before the load balance index falls below the lower threshold.

At each task's arrival, the SA heuristic determines the load balance index. In the worst case, this takes $O(m)$ time. In the next step, the time taken to assign a task to a machine is of order $O(m)$ whether SA uses the MET to perform the mapping or the MCT. Overall, the SA heuristic takes $O(m)$ time irrespective of which heuristic is actually used for mapping the task.

The KPB (k-percent best) heuristic considers only a subset of machines while mapping a task. The subset is formed by picking the $(km/100)$ best machines based on the execution times for the task, where $100/m \leq k \leq 100$. The task is assigned to a machine that provides the earliest completion time in the subset. If $k = 100$, then the KPB heuristic is reduced to the MCT heuristic. If $k = 100/m$, then the KPB heuristic is reduced to the MET heuristic. A "good" value of k maps a task to a machine only within a subset formed from computationally superior machines. The purpose is not as much as matching of the current task to a computationally well-matched machine as it is to avoid putting the current task onto a machine which might be more suitable for some yet-to-arrive tasks. This "foresight" about task heterogeneity lacks in the MCT which might assign a task to a poorly matched machine for an immediate marginal improvement in completion time, possibly depriving some subsequently arriving tasks of that machine, and eventually leading to a larger makespan as compared to the KPB. It should be noted that while both the KPB and SA have elements of the MCT and the MET in their operation, it is only in the KPB that *each* task assignment attempts to optimize objectives of the MCT and the MET simultaneously. However, in cases where a fixed subset of machines is not among the k% best for any task, the KPB will cause much machine idle time compared to the MCT, and can result in much poorer performance.

For each task, $O(m \log m)$ time is spent in ranking the machines for determining the subset of machines to examine. Once the subset of machines is determined, it takes

$O(\frac{km}{100})$ time, i.e., $O(m)$ time to determine the machine assignment. Overall the heuristic takes $O(m \log m)$ time.

The OLB (opportunistic load balancing) heuristic assigns the task to the machine that becomes ready next. It does not consider the execution time of the task when mapping it onto a machine. If multiple machines become ready at the same time, then one machine is arbitrarily chosen.

The complexity of the OLB heuristic is dependent on the implementation. In the implementation considered here, the mapper may need to examine all m machines to find the machine that becomes ready next. Therefore, it takes $O(m)$ to find the assignment. Other implementations may require idle machines to assign tasks to themselves by accessing a shared global queue of tasks [21].

4.3. Batch mode mapping heuristics

In the batch mode heuristics, meta-tasks are mapped after predefined intervals. These intervals are defined in this study using one of the two strategies proposed below.

The regular time interval strategy maps the meta-tasks at regular intervals of time except when all machines are busy. When all machines are busy, all scheduled mapping events that precede the one before the expected ready time of the machine that finishes earliest are canceled.

The fixed count strategy maps a meta-task M_i as soon as one of the following two mutually exclusive conditions are met: (a) an arriving task makes $| M_i |$ larger than or equal to a predetermined arbitrary number κ, or (b) all tasks have arrived, and a task completes while the number of tasks which yet have to begin is larger than or equal to κ. In this strategy, the length of the mapping intervals will depend on the arrival rate and the completion rate. The possibility of machines being idle while waiting for the next mapping event will depend on the arrival rate, completion rate, m, and κ.

The batch mode heuristics considered in this study are discussed in the paragraphs below. The complexity analysis performed for these heuristics considers a single mapping event. In the complexity analysis, the meta-task size is assumed to be equal to the average of meta-task sizes at all actually performed mapping events. Let the average meta-task size be S.

The Min-min heuristic shown in Figure 1 is from SmartNet [6]. In Figure 1, let r_j denote the expected time machine m_j will become ready to execute a task after finishing the execution of all tasks assigned to it at that point in time. First the c_{ij} entries are computed using the e_{ij} and r_j values. For each task t_i the machine that gives the earliest expected completion time is determined by scanning the rows of the c matrix. The task t_k that has the minimum earliest expected completion time is determined and then assigned to the corresponding machine. The matrix c and vector r are updated and the above process is repeated with tasks that have not yet been assigned a machine.

Min-min begins by scheduling the tasks that change the expected machine ready time status by the least amount that any assignment could. If tasks t_i and t_k are contending for a particular machine m_j, then Min-min assigns m_j to the task (say t_i) that will change the ready time of m_j less. This increases the probability that t_k will still have its earliest completion time on m_j, and shall be assigned to it. Because at $t = 0$, the machine which finishes a task earliest is also the one that executes it fastest, and from thereon Min-min heuristic changes machine ready time status by the least amount for every assignment, the percentage of tasks assigned their first choice (on basis of expected execution time) is likely to be higher in Min-min than with the other batch mode heuristics described in this section. The expectation is that a smaller makespan can be obtained if a larger number of tasks is assigned to the machines that not only complete them earliest but also execute them fastest.

```
(1) for all tasks t_i in meta-task M_v (in an arbitrary order)
(2)     for all machines m_j (in a fixed arbitrary order)
(3)         c_ij = e_ij + r_j
(4) do until all tasks in M_v are mapped
(5)     for each task in M_v find the earliest completion
            time and the machine that obtains it
(6)     find the task t_k with the minimum earliest
            completion time
(7)     assign task t_k to the machine m_l that gives the
(8)         earliest completion time
(9)     delete task t_k from M_v
(10)    update r_l
(11)    update c_il for all i
(12) enddo
```

Figure 1. The Min-min heuristic.

The initialization of the c matrix in Line (1) to Line (3) takes $O(Sm)$ time. The **do** loop of the Min-min heuristic is repeated S times and each iteration takes $O(Sm)$ time. Therefore, the heuristic takes $O(S^2m)$ time.

The Max-min heuristic is similar to the Min-min heuristic given in Figure 1. It is also from SmartNet [6]. Once the machine that provides the earliest completion time is found for every task, the task t_k that has the maximum earliest completion time is determined and then assigned to the corresponding machine. The matrix c and vector r are updated and the above process is repeated with tasks that have not yet been assigned a machine. The Max-min heuristic has the same complexity as the Min-min heuristic.

The Max-min is likely to do better than the Min-min heuristic in the cases where we have many more shorter tasks than the long tasks. For example, if there is only one long task, Max-min will execute many short tasks concurrently with the long task. The resulting makespan might just be determined by the execution time of the long task

in these cases. Min-min, however, first finishes the shorter tasks (which may be more or less evenly distributed over the machines) and then executes the long task, increasing the makespan.

The Sufferage heuristic is based on the idea that better mappings can be generated by assigning a machine to a task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it. Let the sufferage value of a task t_i be the difference between its second earliest completion time (on some machine m_y) and its earliest completion time (on some machine m_x). That is, using m_x will result in the best completion time for t_i, and using m_y the second best.

Figure 2 shows the Sufferage heuristic. The initialization phase in Lines (1) to (3) is similar to the ones in the Min-min and Max-min heuristics. Initially all machines are marked unassigned. In each iteration of the **for** loop in Lines (6) to (14), pick arbitrarily a task t_k from the meta-task. Find the machine m_j that gives the earliest completion time for task t_k, and tentatively assign m_j to t_k if m_j is unassigned. Mark m_j as assigned, and remove t_k from meta-task. If, however, machine m_j has been previously assigned to a task t_i, choose from t_i and t_k the task that has the higher sufferage value, assign m_j to the chosen task, and remove the chosen task from the meta-task. The unchosen task will not be considered again for this execution of the **for** statement, but shall be considered for the next iteration of the **do** loop beginning on Line (4). When all the iterations of the **for** loop are completed (i.e., when one execution of the **for** statement is completed), update the machine ready time of the each machine assigned a new task. Perform the next iteration of the **do** loop beginning on Line (4) until all tasks have been mapped.

Table 1 shows a scenario in which the Sufferage will outperform the Min-min. Table 1 shows the expected execution time values for four tasks on four machines (all initially idle). In this particular case, the Min-min heuristic gives a makespan of 9.3 and the Sufferage heuristic gives a makespan of 7.8. Figure 3 gives a pictorial representation of the assignments made for the case in Table 1.

From the pseudo code given in Figure 2, it can be observed that first execution of the **for** statement on Line (6) takes $O(Sm)$ time. The number of task assignments made in one execution of this **for** statement depends on the total number of machines in the HC suite, the number of machines that are being contended for among different tasks, and the number of tasks in the meta-task being mapped. In the worst case, only one task assignment will be made in each execution of the **for** statement. Then meta-task size will decrease by one at each **for** statement execution. The outer **do** loop will be iterated S times to map the whole meta-task. Therefore, in the worst case, the time $T(S)$ taken

	m_0	m_1	m_2	m_3
t_0	4	4.8	13.4	5
t_1	5	8.2	8.8	8.9
t_2	5.5	6.8	9.4	9.3
t_3	5.2	6	7.8	10.8

Table 1. An example expected execution time matrix that illustrates the situation where the Sufferage heuristic outperforms the Min-min heuristic.

to map a meta-task of size S will be

$$T(S) = Sm + (S-1)m + (S-2)m + \cdots + m$$

$$T(S) = O(S^2 m)$$

In the best case, there are as many machines as there are tasks in the meta-task, and there is no contention among the tasks. Then all the task are assigned in the first execution of the **for** statement so that $T(S) = O(Sm)$. Let $\underline{\omega}$ be a number quantifying the extent of contention among the tasks for the different machines. The running time of Sufferage heuristic can then be given as $O(\omega Sm)$ time, where $1 \leq \omega \leq S$. It can be seen that ω is equal to S in the worst case, and is 1 in the best case; these values of ω are numerically equal to the number of iterations of the **do** loop on Line (4).

```
(1)  for all tasks t_k in meta-task M_v (in an arbitrary order)
(2)      for all machines m_j (in a fixed arbitrary order)
(3)          c_kj = e_kj + r_j
(4)  do until all tasks in M_v are mapped
(5)      mark all machines as unassigned
(6)      for each task t_k in M_v (in an arbitrary order)
(7)          find machine m_j that gives the earliest
                 completion time
(8)          sufferage value = second earliest completion
                 time - earliest completion time
(9)          if machine m_j is unassigned
(10)             assign t_k to machine m_j, delete t_k
                 from M_v, mark m_j assigned
(11)         else
(12)             if sufferage value of task t_i already
                 assigned to m_j is less than the
                 sufferage value of task t_k
(13)                 unassign t_i, add t_i back to M_v,
                     assign t_k to machine m_j,
                     delete t_k from M_v
(14)     endfor
(15)     update the vector r based on the tasks that
             were assigned to the machines
(16)     update the c matrix
(17) enddo
```

Figure 2. The Sufferage heuristic.

The batch mode heuristics can cause some tasks to be starved of machines. Let H_i be a subset of meta-task M_i consisting of tasks that were mapped (as part of M_i) at the mapping event i at time τ_i but did not begin execution by the next mapping event at τ_{i+1}. H_i is the subset of M_i that is included in M_{i+1}. Due to the expected heterogeneous nature of the tasks, the meta-task M_{i+1} may be so mapped that some or all of the tasks arriving between τ_i and τ_{i+1} may begin executing before the tasks in set H_i do. It is possible that some or all of the tasks in H_i may be included in H_{i+1}. This probability increases as the number of new tasks arriving between τ_i and τ_{i+1} increases. In general, some tasks may be remapped at each successive mapping event without actually beginning execution (i.e., the task is starving for a machine).

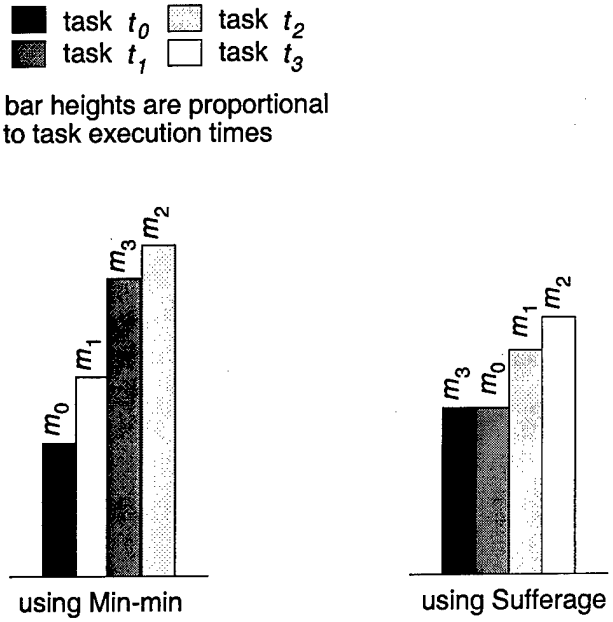


Figure 3. An example scenario (based on Table 1) where the Sufferage gives a shorter makespan than the Min-min.

To reduce starvation, aging schemes are implemented. The age of a task is set to zero when it is mapped for the first time and incremented by one each time the task is remapped. Let σ be a constant that can be adjusted empirically to change the extent to which aging affects the operation of the heuristic. An aging factor, $\zeta = (1 + \text{age}/\sigma)$, is then computed for each task. For the experiments in this study, σ is set to 10. The aging factor is used to enhance the probability of an "older" task beginning before the tasks that would otherwise begin first. In the Min-min heuristic, for each task, the completion time obtained in Line (5) of Figure 1 is multiplied by the corresponding value for $\frac{1}{\zeta}$. As the age of a task increases, its age-compensated expected completion time (i.e., one used to determine the mapping) gets increasingly smaller than its original expected completion time. This increases its probability of being selected in Line (6) in Figure 1.

Similarly, for the Max-min heuristic, the completion time of a task is multiplied by ζ. In the Sufferage heuristic, the sufferage value computed in Line (8) in Figure 2 is multiplied by ζ.

5. Simulation Procedure

The mappings are simulated using a discrete event simulator. The task arrivals are modeled by a Poisson random process. The simulator contains an ETC (expected time to compute) matrix that contains the expected execution times of a task on all machines, for all the tasks that can arrive for service. The ETC matrix entries used in the simulation studies represent the e_{ij} values that the heuristic would use in its operation. The actual execution time of a task can be different from the value given by the ETC matrix. This variation is modeled by generating a simulated actual execution time for each task by sampling a truncated Gaussian probability density function with variance equal to three times the expected execution time of the task and mean equal to the expected execution time of the task [2, 16]. If the sampling results in a negative value, the value is discarded and the same probability density function is sampled again. This process is repeated until a positive value is returned by the sampling process.

In an ETC matrix, the numbers along a row indicate the execution times of the corresponding task on different machines. The average variation along the rows is referred to as the machine heterogeneity [2]. Similarly, the average variation along the columns is referred to as the task heterogeneity [2]. One classification of heterogeneity is to divide it into high heterogeneity and low heterogeneity. Based on the above idea, four categories were proposed for the ETC matrix in [2]: (a) high task heterogeneity and high machine heterogeneity (HiHi), (b) high task heterogeneity and low machine heterogeneity (HiLo), (c) low task heterogeneity and high machine heterogeneity (LoHi), and (d) low task heterogeneity and low machine heterogeneity (LoLo). The ETC matrix can be further classified into two classes, consistent and inconsistent, which are orthogonal to the previous classifications. For a consistent ETC matrix, if machine m_x has a lower execution time than machine m_y for task t_k, then the same is true for any task t_i. The ETC matrices that are not consistent are inconsistent ETC matrices. In addition to the consistent and inconsistent classes, a semi-consistent class could also be defined. A semi-consistent ETC matrix is characterized by a consistent sub-matrix. In the semi-consistent ETC matrices used here, 50% of the tasks and 25% of the machines define a consistent sub-matrix. Furthermore, it is assumed that for a

particular task the execution times that fall within the consistent sub-matrix are smaller than those that fall out. This assumption is justified because the machines that perform consistently better than the others for some tasks are more likely to be very much faster for those tasks than very much slower.

Let an ETC matrix have t_{max} rows and m_{max} columns. Random ETC matrices that belong to the different categories are generated in the following manner:

1. Let Γ_t be an arbitrary constant quantifying task heterogeneity, being smaller for low task heterogeneity. Let N_t be a number picked from the uniform random distribution with range $[1, \Gamma_t]$.
2. Let Γ_m be an arbitrary constant quantifying machine heterogeneity, being smaller for low machine heterogeneity. Let N_m be a number picked from the uniform random distribution with range $[1, \Gamma_m]$.
3. Sample N_t t_{max} times to get a vector $q[0..(t_{max}-1)]$.
4. Generate the ETC matrix, $e[0..(t_{max}-1), 0..(m_{max}-1)]$ by the following algorithm:

```
for t_i from 0 to (t_max − 1)
    for m_j from 0 to (m_max − 1)
        pick a new value for N_m
        e[i, j] = q[i] * N_m.
    endfor
endfor
```

From the raw ETC matrix generated above, a semi-consistent matrix could be generated by sorting the execution times for a random subset of the tasks on a random subset of machines. An inconsistent ETC matrix could be obtained simply by leaving the raw ETC matrix as such. Consistent ETC matrices were not considered in this study because they are least likely to arise in the current intended MSHN environment.

In the experiments described here, the values of Γ_t for low and high task heterogeneities are 1000 and 3000, respectively. The values of Γ_m for low and high machine heterogeneities are 10 and 100, respectively. These heterogeneous ranges are based on one type of expected environment for MSHN.

6. Experimental Results and Discussion

6.1. Overview

The experimental evaluation of the heuristics is performed in three parts. In the first part, the on-line mode heuristics are compared using various metrics. The second part involves a comparison of the batch mode heuristics. The third part is the comparison of the batch mode and the on-line mode heuristics. Unless stated otherwise, the following are valid for the experiments described here. The number of machines is held constant at 20, and the experiments are performed for $|K| = \{1000, 2000\}$. All heuristics are evaluated in a HiHi heterogeneity environment, both for the inconsistent and the semi-consistent cases, because these correspond to some of the currently expected MSHN environments. A Poisson distribution is used to generate the task arrivals. For each value of $|K|$, tasks are mapped under two different arrival rates, λ_h and λ_l, such that $\lambda_h > \lambda_l$. The value of λ_h is chosen empirically to be high enough to allow at most 50% tasks to complete when the last task in the set arrives. Similarly, λ_l is chosen to be low enough to allow at least 90% of the tasks to complete when the last task in the set arrives. The MCT heuristic is used in this standardization. Unless otherwise stated, the task arrival rate is set to λ_h. λ_l is more likely to represent an HC system where the task arrival is characterized by little burstiness; no particular group of tasks arrives in a much shorter span of time than some other group having same number of tasks. λ_h is supposed to characterize the arrivals in an HC system where a large group of tasks arrives in a much shorter time than some other group having same number of tasks; e.g., in this case a burst of $|K|$ tasks.

Example comparisons are discussed in Subsections 6.2 to 6.4. Each data point in the comparison charts is an average over 50 trials, where for each trial the simulated actual task execution times are chosen independently. More general conclusions about the heuristics' performance is in Section 7. Comparisons for a larger set of performance metrics are given in [13].

6.2. Comparisons of the on-line mode heuristics

Unless otherwise stated, the on-line mode heuristics are investigated under the following conditions. In the KPB heuristic, k is equal to 20%. This particular value of k was found to give the lowest makespan for the KPB heuristic under the conditions of the experiments. For the SA, the lower threshold and the upper threshold for the load balance index are 0.6 and 0.9, respectively. Once again these values were found to give optimum values of makespan for the SA.

In Figure 4, on-line mode heuristics are compared based on makespan for inconsistent HiHi heterogeneity. From Figure 4, it can be noted that the KPB heuristic completes the execution of the last finishing task earlier than the other heuristics (however, it is only slightly better than the MCT). For $k = 20\%$ and $m = 20$, the KPB heuristic forces a task to choose a machine from a subset of four machines. These four machines have the lowest execution times for the given task. The chosen machine would give the smallest completion time as compared to other machines in the set.

Figure 5 compares the on-line mode heuristics using average sharing penalty. Once again, the KPB heuristic per-

forms best. However, the margin of improvement is smaller than that for the makespan. It is evident that the KPB provides maximum throughput (system oriented performance metric) and minimum average sharing penalty (application oriented performance metric).

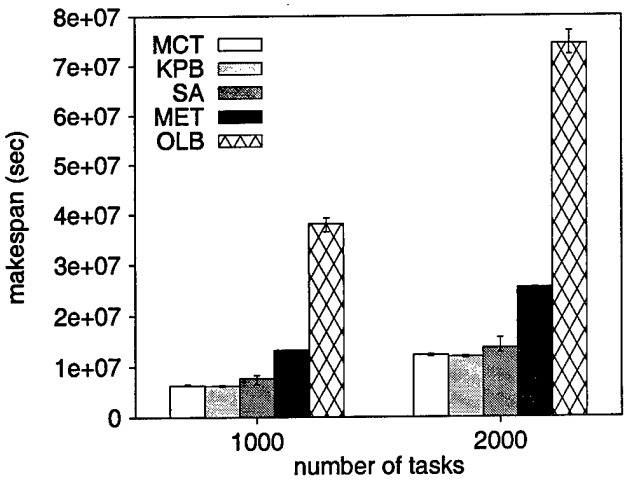


Figure 4. Makespan for the on-line heuristics for inconsistent HiHi heterogeneity.

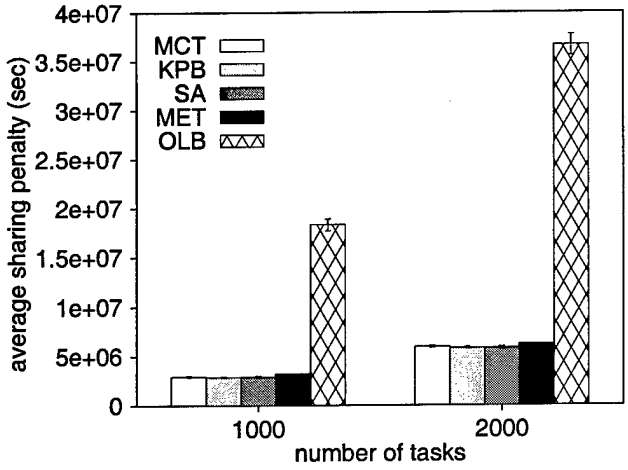


Figure 5. Average sharing penalty of the on-line heuristics for inconsistent HiHi heterogeneity.

Figure 6 compares the makespans of the different on-line heuristics for semi-consistent HiHi ETC matrices. Figure 7 compares the average sharing penalties of the different on-line heuristics. As shown in Figures 4 and 6 the relative performance of the different on-line heuristics is impacted by the degree of consistency of the ETC matrices.

For the semi-consistent type of heterogeneity, machines within a particular subset perform tasks that lie within a particular subset faster than other machines. From Figure 6, it can be observed that for semi-consistent ETC matrices, the MET heuristic performs the worst. For the semi-consistent matrices used in these simulations, the MET heuristic maps half of the tasks to the same machine, considerably increasing the load imbalance. Although the KPB also considers only the fastest four machines for each task for the particular value of k used here (which happen to be the same four machines for half of the tasks), the performance does not differ much from the inconsistent HiHi case. Additional experiments have shown that the KPB performance is quite insensitive to values of k as long as k is larger than the minimum value (where the KPB heuristic is reduced to the MET heuristic). For example, when k is doubled from its minimum value of 5, the makespan decreases by a factor of about 5. However a further doubling of k brings down the makespan by a factor of only about 1.2.

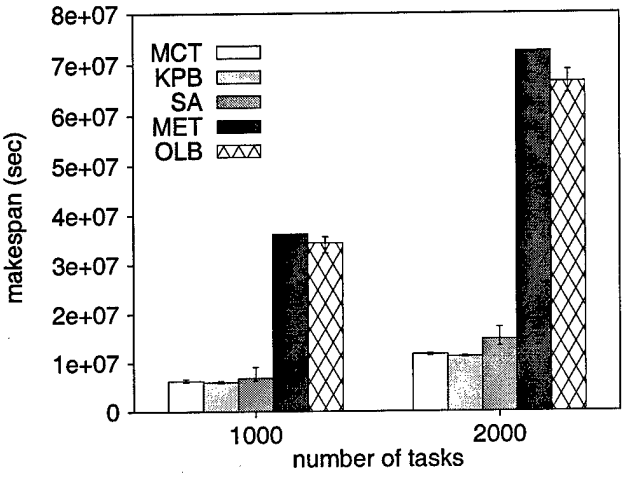


Figure 6. Makespan of the on-line heuristics for semi-consistent HiHi heterogeneity.

6.3. Comparisons of the batch mode heuristics

Figures 8 and 9 compare the batch mode heuristics based on makespan and average sharing penalty, respectively. In these comparisons, unless otherwise stated, the regular time interval strategy is employed to schedule meta-task mapping events. The time interval is set to 10 seconds. This value was empirically found to optimize makespan over other values. From Figure 8, it can be noted that the Sufferage heuristic outperforms the Min-min and the Max-min heuristics based on makespan (although, it is only slightly better than the Min-min). However, for average sharing penalty, the Min-min heuristic outperforms the other heuristics (Figure 9). The Sufferage heuristic considers the "loss" in completion time of a task if it is not assigned to its first choice, in making the mapping decisions. By assigning their first choice machines to the tasks that have the highest sufferage values among all contending tasks, the Sufferage heuristic reduces the overall completion time.

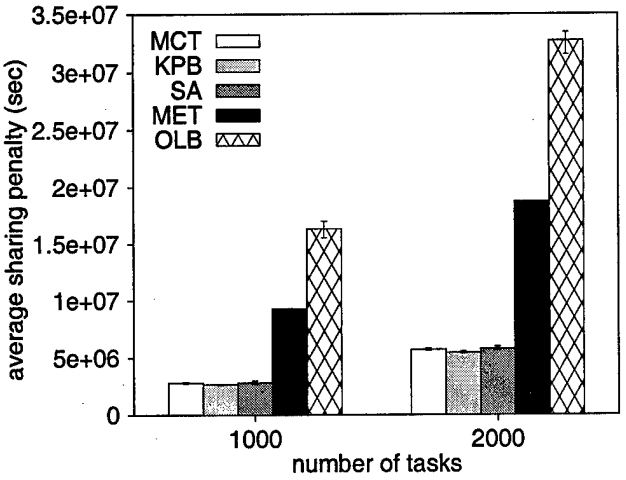


Figure 7. Average sharing penalty of the on-line heuristics for semi-consistent HiHi heterogeneity.

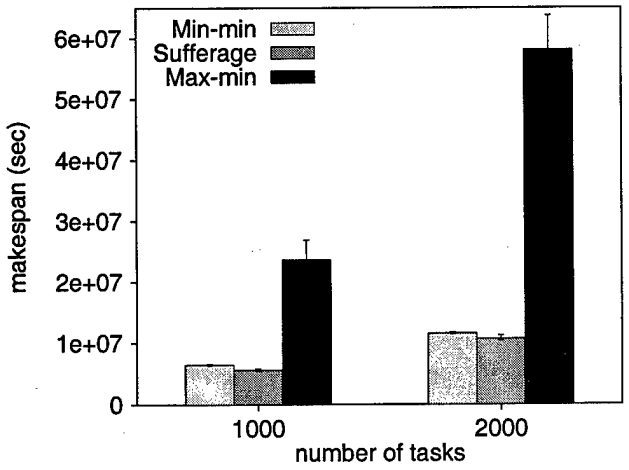


Figure 8. Makespan of the batch heuristics for the regular time interval strategy and inconsistent HiHi heterogeneity.

Furthermore, it can be noted that the makespan given by the Max-min is much larger than the makespans obtained by the other two heuristics. Using reasoning similar to that given in Subsection 4.3 for explaining better expected performance for the Min-min, it can be seen that the Max-min assignments change the machine ready time status by a larger amount than the Min-min assignments do. (The Sufferage also does not necessarily schedule the tasks that finish later first.) If tasks t_i and t_k are contending for a particular machine m_j, then the Max-min assigns m_j to the task (say t_i) that will increase the ready time of m_j more. This decreases the probability that t_k will still have its earliest completion time on m_j and shall be assigned to it. In general, the percentage of tasks assigned their first

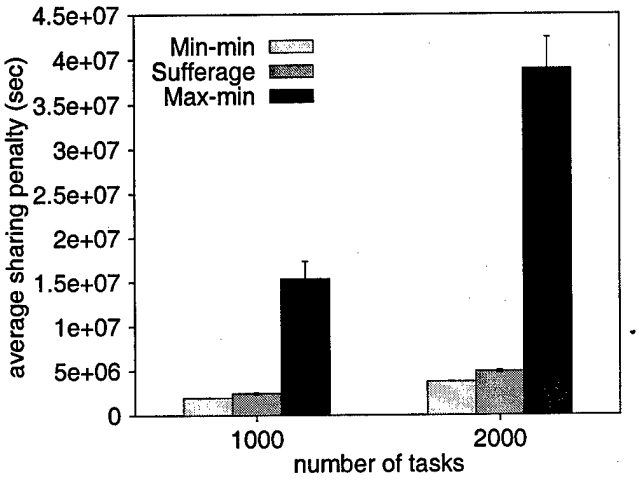


Figure 9. Average sharing penalty of the batch heuristics for the regular time interval strategy and inconsistent HiHi heterogeneity.

choice is likely to be lower for the Max-min than for other batch mode heuristics. It might be expected that a larger makespan will result if a larger number of tasks is assigned to the machines that do not have the best execution times for those tasks.

Figure 10 compares the makespan of the batch mode heuristics for semi-consistent HiHi heterogeneity. The comparison of the same heuristics for the same parameters is shown in Figure 11 with respect to average sharing penalty. Results for both average sharing penalty and makespan for semi-consistent HiHi are similar to those for inconsistent HiHi.

The impact of aging on batch mode heuristics is shown in Figures 12 and 13. From Figures 12 and 13, three observations are in order. First, the Max-min heuristic benefits most from the aging scheme. Second, the makespan and the average sharing penalty given by the Sufferage heuristic change negligibly when aging scheme is applied. Third, even though aging schemes are meant to reduce starvation of tasks (as gauged by average sharing penalty), they also reduce the makespan.

The fact that the Max-min benefits most from the aging scheme can be explained using the reasoning given in the discussion on starvation in Subsection 4.3. The larger the number (say N_{new}) of newly arriving tasks between the mapping events τ_i and τ_{i+1}, the larger the probability that some of the tasks mapped at mapping event τ_i, or earlier, will be starved (due to more competing tasks). The Max-min heuristic schedules tasks that finish later first. As mapping events are not scheduled if machines are busy, two successive mapping events in the Max-min are likely to be separated by a larger time duration than those in the Sufferage or the Min-min. The value of N_{new} is therefore likely to

be larger in the Max-min schedules, and starvation is more likely to occur. Consequently, aging schemes would make greater difference to the Max-min schedules: the tasks that finish sooner are much more likely to be scheduled before the tasks that finish later in the Max-min with aging than in the Max-min without aging. In contrast to the Max-min (or the Min-min) operation, the Sufferage heuristic optimizes a machine assignment only over the tasks that are contending for that particular machine. This reduces the probability of competition between the "older" tasks and the new arrivals, which in turn reduces the need for an aging scheme, or the improvement in schedule in case aging is implemented.

Figures 14, 15, 16, and 17 show the results of repeating the above experiments with a batch count mapping strategy for a batch size of 40. This particular batch size was found to give an optimum value of the makespan. Figure 14 compares regular time interval strategy and fixed count strategy on the basis of makespans given by different heuristics for inconsistent HiHi heterogeneity. In Figure 15, the average sharing penalties of the same heuristics for the same parameters are compared. It can be seen that the fixed count approach gives essentially the same results for the Min-min and the Sufferage heuristics. The Max-min heuristic, however, benefits considerably from the fixed count approach; makespan drops to about 60% for $| K |= 1000$, and to about 50% for $| K |= 2000$ as compared to the makespan given by the regular time interval strategy. A possible explanation lies in a conceptual element of similarity between the fixed count approach and the aging scheme. A "good" value of κ in fixed count strategy is neither too small to allow only a limited optimization of machine assignment nor too large to subject the tasks carried over from the previous mapping events to a possibly defeating competition with the new or recent arrivals. Figures 16 and 17 show the makespan and the average sharing penalty given for the semi-consistent case. These results show that, for the Sufferage and the Min-min, the regular time interval approach gives slightly better results than the fixed count approach. For the Max-min, however, the fixed count approach gives better performance.

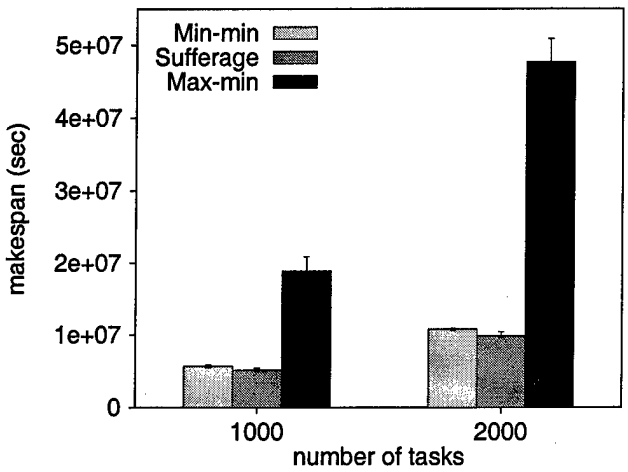


Figure 10. Makespan of the batch heuristics for the regular time interval strategy and semi-consistent HiHi heterogeneity.

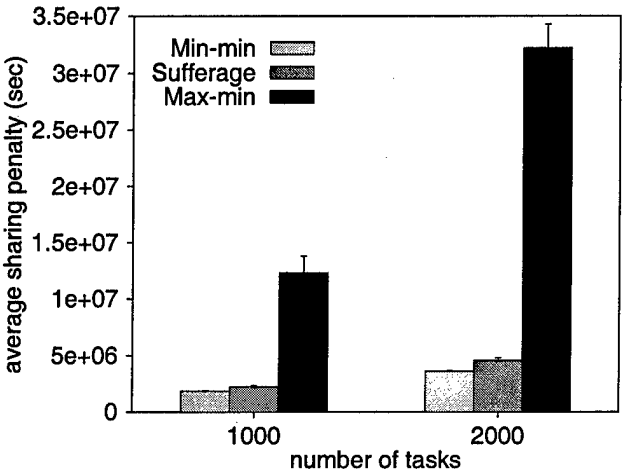


Figure 11. Average sharing penalty of the batch heuristics for the regular time interval strategy and semi-consistent HiHi heterogeneity.

6.4. Comparing on-line and batch heuristics

In Figure 18, two on-line mode heuristics, the MCT and the KPB, are compared with two batch mode heuristics, the Min-min and the Sufferage. The comparison is performed with Poisson arrival rate set to λ_h. It can be noted that for the higher arrival rate and larger $| K |$, batch heuristics are superior to on-line heuristics. This is because the number of tasks waiting to begin execution is likely to be larger in above circumstances than in any other, which in turn means that rescheduling is likely to improve many more mappings in such a system. The on-line heuristics consider only one task when they try to optimize machine assignment, and do not reschedule. Recall that the mapping heuristics use a combination of expected and actual task execution times to compute machine ready times. The on-line heuristics are likely to approach the performance of the batch ones at low task arrival rates, because then both classes of heuristics have comparable information about the actual execution times of the tasks. For example, at a certain low arrival rate, the 100-th arriving task might find that 70 previously arrived tasks have completed. At a higher arrival rate, only 20 tasks might have completed when the 100-th task arrived. The above observation is borne out in Figure 19, which shows that the relative performance difference between on-line and batch heuristics decreases with a decrease in arrival rate. Given the observation that the KPB and the Sufferage per-

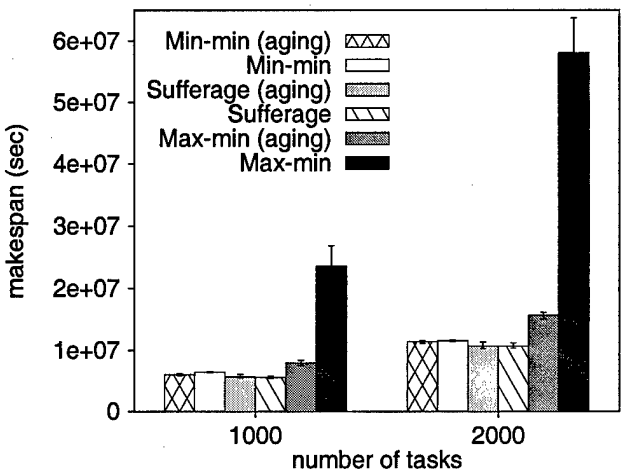


Figure 12. Makespan for the batch heuristics for the regular time interval strategy with and without aging for inconsistent HiHi heterogeneity.

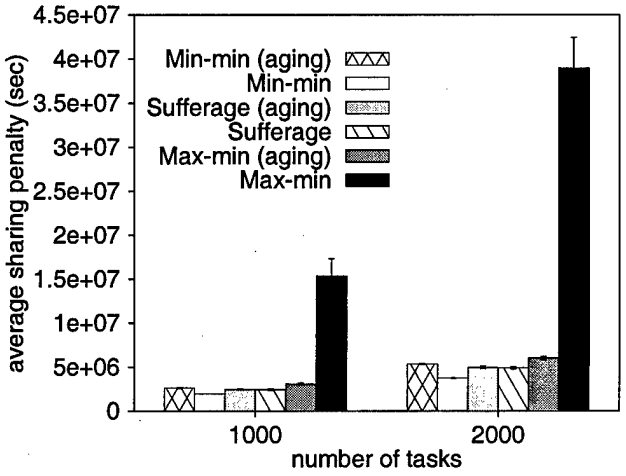


Figure 13. Average sharing penalty of the batch heuristics for the regular time interval strategy with and without aging for inconsistent HiHi heterogeneity.

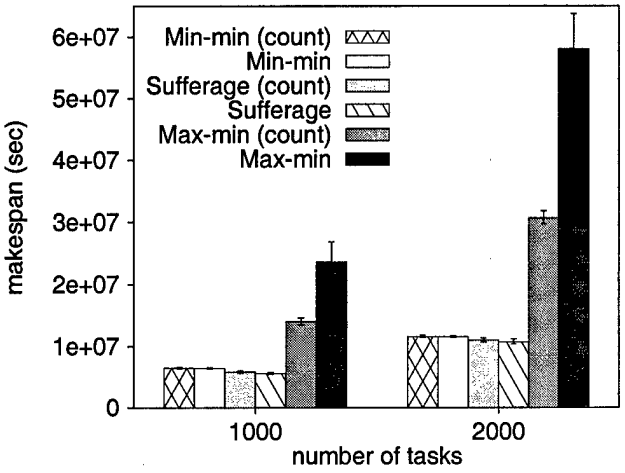


Figure 14. Comparison of the makespans given by the fixed count mapping strategy and the regular time interval strategy for inconsistent HiHi heterogeneity.

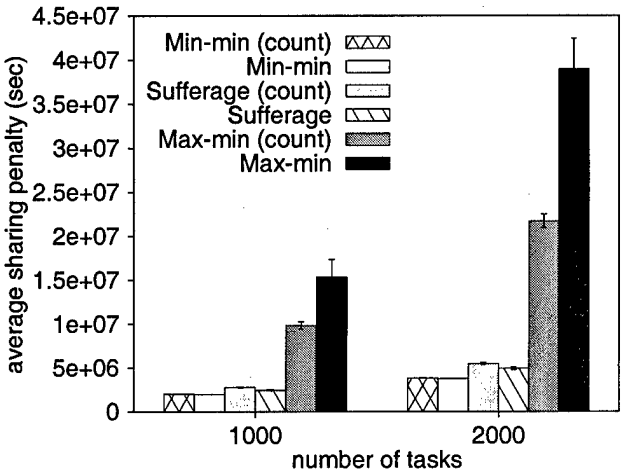


Figure 15. Comparison of the average sharing penalty given by the fixed count mapping strategy and the regular time interval strategy for inconsistent HiHi heterogeneity.

form almost similarly at this low arrival rate, it might be better to use the KPB heuristic because of its smaller computation time. Moreover, Figures 18 and 19 show that the makespan values for all heuristics are larger for lower arrival rate. This is attributable to the fact that at lower arrival rates, a larger fraction of a task's completion time is determined by its beginning time.

7. Conclusions

New and previously proposed dynamic matching and scheduling heuristics for mapping independent tasks onto HC systems were compared under a variety of simulated computational environments. Five on-line mode heuristics and three batch mode heuristics were studied.

In the on-line mode, for both the semi-consistent and the inconsistent types of HiHi heterogeneity, the KPB heuristic outperformed the other heuristics on all performance metrics (however, the KPB was only slightly better than the MCT). The average sharing penalty gains were smaller than the makespan ones. The KPB can provide good system oriented performance (e.g., minimum makespan) and at the same time provide good application oriented performance (e.g., low average sharing penalty). The relative performance of the OLB and the MET with respect to the makespan reversed when the heterogeneity was changed

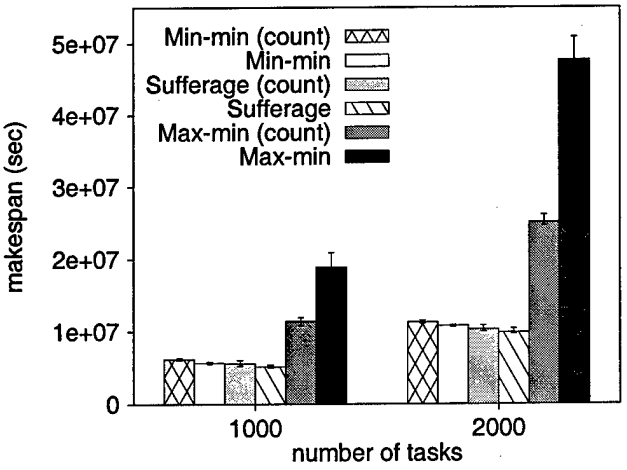


Figure 16. Comparison of the makespan given by the fixed count mapping strategy and the regular time interval strategy for semi-consistent HiHi heterogeneity.

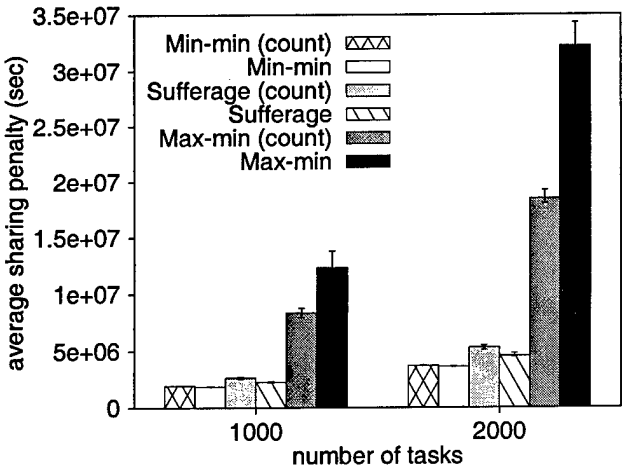


Figure 17. Comparison of the average sharing penalty given by the fixed count mapping strategy and the regular time interval strategy for semi-consistent HiHi heterogeneity.

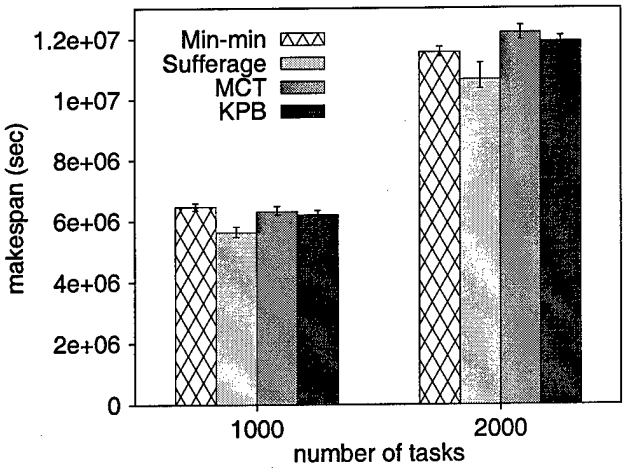


Figure 18. Comparison of the makespan given by batch heuristics (regular time interval strategy) and on-line heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_h.

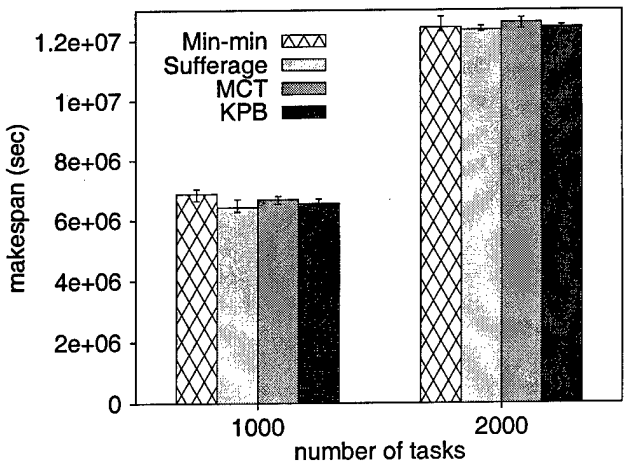


Figure 19. Comparison of the makespan given by batch heuristics (regular time interval strategy) and on-line heuristics for inconsistent HiHi heterogeneity and an arrival rate of λ_l.

from the semi-consistent to the inconsistent. The OLB did better than the MET for the semi-consistent case.

In the batch mode, for the semi-consistent and the inconsistent types of HiHi heterogeneity, the Min-min heuristic outperformed the Sufferage and Max-min heuristics in the average sharing penalty. However, the Sufferage performed the best with respect to makespan for both the semi-consistent and the inconsistent types of HiHi heterogeneity (though, the Sufferage was only slightly better than the Min-min).

The batch heuristics are likely to give a smaller makespan than the on-line ones for large $| K |$ and high task arrival rate. For smaller values of $| K |$ and lower task arrival rates, the improvement in makespan offered by batch heuristics is likely to be nominal.

This study quantifies how the relative performance of these dynamic mapping heuristics depends on (a) the consistency property of the ETC matrix, (b) the requirement to optimize system oriented or application oriented performance metrics (e.g., optimizing makespan versus optimizing average sharing penalty), and (c) the arrival rate of the tasks. Thus, the choice of the heuristic which is best to use will be a function of such factors. Therefore, it is important to include a set of heuristics in a resource

management system for HC environments, and then use the heuristic that is most appropriate for a given situation (as will be done in the Scheduling Advisor for MSHN).

Acknowledgments: The authors thank Taylor Kidd, Surjamukhi Chatterjea, and Tracy D. Braun for their valuable comments and suggestions.

References

[1] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predications," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 79–87.

[2] R. Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance*, Master's thesis, Department of Computer Science, Naval Postgraduate School, 1997 (D. Hensgen, advisor).

[3] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems," *IEEE Workshop on Advances in Parallel and Distributed Systems*, Oct. 1998, pp. 330–335 (included in the proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, 1998).

[4] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, R. F. Freund, and D. Hensgen, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, Apr. 1999, to appear.

[5] I. Foster and C. Kesselman (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Fransisco, CA, 1999.

[6] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 184–199.

[7] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 78–86.

[8] D. Hensgen, T. Kidd, M. C. Schnaidt, D. St. John, H. J. Siegel, T. D. Braun, M. Maheshwaran, S. Ali, J-K. Kim, C. Irvine, T. Levin, R. Wright, R. F. Freund, M. Godfrey, A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, "An overview of MSHN: A Management System for Heterogeneous Networks," *8th IEEE Heterogeneous Computing Workshop (HCW '99)*, Apr. 1999, to appear.

[9] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, Vol. 24, No. 2, Apr. 1977, pp. 280–289.

[10] M. A. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998, pp. 70–78.

[11] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, Vol. 6, No. 3, July-Sep. 1998, pp. 42–51.

[12] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," *4th IEEE Heterogeneous Computing Workshop (HCW '95)*, Apr. 1995, pp. 30–34.

[13] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, *A Comparison of Dynamic Strategies for Mapping a Class of Independent Tasks onto Heterogeneous Computing Systems*, Technical Report, School of Electrical and Computer Engineering, Purdue University, in preparation, 1999.

[14] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," in *Encyclopedia of Electrical and Electronics Engineering*, J. G. Webster, ed., John Wiley, New York, NY, scheduled to appear in 1999.

[15] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Adaptive load sharing in heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, Vol. 9, No. 4, Aug. 1990, pp. 331–346.

[16] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, NY, 1984.

[17] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[18] H. G. Rotithor, "Taxonomy of dynamic task scheduling schemes in distributed computing systems," *IEE Proceedings on Computer and Digital Techniques*, Vol. 141, No. 1, Jan. 1994, pp. 1–10.

[19] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, Apr. 1996, pp. 86–97.

[20] V. Suresh and D. Chaudhuri, "Dynamic rescheduling–A survey of research," *International Journal of Pro-*

duction Economics, Vol. 32, No. 1, Aug. 1993, pp. 53–63.

[21] P. Tang, P. C. Yew, and C. Zhu, "Impact of self-scheduling on performance of multiprocessor systems," *3rd International Conference on Supercomputing*, July 1988, pp. 593–603.

Biographies

Muthucumaru Maheswaran is an Assistant Professor in the Department of Computer Science at the University of Manitoba, Canada. In 1990, he received a BSc degree in electrical and electronic engineering from the University of Peradeniya, Sri Lanka. He received an MSEE degree in 1994 and a PhD degree in 1998, both from the School of Electrical and Computer Engineering at Purdue University. He held a Fulbright scholarship during his tenure as an MSEE student at Purdue University. His research interests include computer architecture, distributed computing, heterogeneous computing, Internet and world wide web systems, metacomputing, mobile programs, network computing, parallel computing, resource management systems for metacomputing, and scientific computing. He has authored or coauthored 15 technical papers in these and related areas. He is a member of the Eta Kappa Nu honorary society.

Shoukat Ali is pursuing an MSEE degree from the School of Electrical and Computer Engineering at Purdue University, where he is currently a Research Assistant. His main research topic is dynamic mapping of meta-tasks in heterogeneous computing systems. He has held teaching positions at Aitchison College and Keynesian Institute of Management and Sciences, both in Lahore, Pakistan. He was also a Teaching Assistant at Purdue. Shoukat received his BS degree in electrical and electronic engineering from the University of Engineering and Technology, Lahore, Pakistan in 1996. His research interests include computer architecture, parallel computing, and heterogeneous computing.

Howard Jay Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is a Fellow of the IEEE and a Fellow of the ACM. He received BS degrees in both electrical engineering and management from MIT, and the MA, MSE, and PhD degrees from the Department of Electrical Engineering and Computer Science at Princeton University. Prof. Siegel has coauthored over 250 technical papers, has coedited seven volumes, and wrote the book *Interconnection Networks for Large-Scale Parallel Processing*. He was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing*, and was on the Editorial Boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He was Program Chair/Co-Chair of three conferences, General Chair/Co-Chair of four conferences, and Chair/Co-Chair of four workshops. He is an international keynote speaker and tutorial lecturer, and a consultant for government and industry.

Debra Hensgen is an Associate Professor in the Computer Science Department at The Naval Postgraduate School. She received her PhD in the area of Distributed Operating Systems from the University of Kentucky. She is currently a Principal Investigator of the DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) and a co-investigator of the DARPA-sponsored Server and Active Agent Management (SAAM) Next Generation Internet project. Her areas of interest include active modeling in resource management systems, network re-routing to preserve quality of service guarantees, visualization tools for performance debugging of parallel and distributed systems, and methods for aggregating sensor information. She has published numerous papers concerning her contributions to the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems.

Richard F. Freund is a founder and CEO of NOEMIX, a San Diego based startup to commercialize distributed computing technology. Freund is also one of the early pioneers in the field of distributed computing, in which he has written or co-authored a number of papers. In addition he is a founder of the Heterogeneous Computing Workshop, held each year in conjunction with IPPS/SPDP. Freund won a Meritorious Civilian Service Award during his former career as a government scientist.

Session II

Design Tools

Chair

Ishfaq Ahmad
Hong Kong University of Science and Technology

An On-Line Performance Visualization Technology*

Aleksandar M. Bakić, Matt W. Mutka
Michigan State University
Department of Computer
Science and Engineering
3115 Engineering Building
East Lansing, Michigan 48824
{bakicale,mutka}@cse.msu.edu

Diane T. Rover
Michigan State University
Department of Electrical
and Computer Engineering
2120 Engineering Building
East Lansing, Michigan 48824
rover@egr.msu.edu

Abstract

We present a new software technology for on-line performance analysis and visualization of complex parallel and distributed systems. Often heterogeneous, these systems need capabilities for flexible integration and configuration of performance analysis and visualization. Our technology is based on an object-oriented framework for rapid prototyping and development of distributable visual objects. *The visual objects consist of two levels, a platform/device-specific low level, and an analysis- and visualization-specific high level. We have developed a very high-level, markup language, called VOML, and a compiler for component-based development of high-level visual objects. The VOML is based on a software architecture for on-line event processing and performance visualization called EPIRA. The technology lends itself to constructing high-level visual objects from globally distributed component definitions. We present details of the technology and tools used, and show how an example visual object can be rapidly prototyped from several reusable components.*

1 Introduction

Performance analysis and visualization (PAV) tools are crucial components of an effective development cycle, as well as deployment, of parallel and distributed applications. On-line PAV is even becoming necessary for the latter. Since the amount of performance data to be analyzed and visualized increases with the size of a target parallel/distributed application, on-line PAV itself should be distributed. Heterogeneous systems, in addition, need PAV tools that provide flexible integration and configuration support for heterogeneous performance data. Extant generic and library-specific PAV tools for parallel/distributed systems can cover only low-level performance aspects, provided that the target systems fit into their generic schemes and/or use specific libraries, such as PVM [8] and MPI [7]. A wider range of performance aspects, at multiple levels, global and local, are needed to capture and visually explain the behavior of a heterogeneous system.

We have developed a framework for on-line PAV, called PG^{RT} visual objects[1], to address these issues. The framework is object-oriented and easily distributable via middleware software such as CORBA [16] and DCOM [3]. Within it, a visual-object developer can integrate low- and high-level, application-specific PAV. Furthermore, it is based on two visual-object levels for portability and code reuse: a device-dependent low level, and a device-independent high-level. Our goal was also to be able to integrate various sources of off- and on-line performance data. To achieve this flexibility, the visual objects consume performance data in the form of *event records* from an environment. To formalize the design of high-level visual objects, i.e., enforce a structured approach that is less error-prone, we have defined certain rules and a very high-level, component-based specification language, called Visual Object Markup Language (VOML). The language uses Standard Generalized Markup Language (SGML) markup for structuring visual objects, and Scheme scripts for defining PAV semantics.

The use of SGML enables development of a PAV information infrastructure for platform- and tool-independent development of visual objects. It may also facilitate automatic monitoring, analysis, and visualization of globally distributed applications via network-enabled SGML entity managers. The use of Scheme for visual object semantics

*This work was supported in part by DARPA contract No. DABT 63–95–C–0072, NSF grant No. CDA–9529488, and NSF grant No. ASC–9624149.

[1] PG^{RT} is the acronym of our Performance Gateway to Real-Time project.

0-7695-0107-9/99 $10.00 © 1999 IEEE

enables both rapid prototyping of visual objects and customizing VOs for a wide range of platforms via, for example, Scheme-to-C and Scheme-to-Java VM bytecode compilers. That is, a single VOML specification may be used to generate automatically an X library-based visual object and one that runs within a WWW browser.

In Section 2, we describe the visual-object framework in detail, and show an example of successful use for PAV of a distributed multimedia real-time application. A PAV architecture for high-level visual objects, the markup language based on it, and development environment are presented in Section 3. An example of a VOML specification is given in Section 4. We compare our PAV approach to other work in the area in Section 5, and conclude in Section 6.

2 Visual Object Architecture

The Visual Object (VO) architecture identifies two main software layers apparent in the majority of extant PAV tools, and represents them as two classes: the *high-level VO (HLVO) class* and the *low-level VO (LLVO) class*. In general, the responsibility of an HLVO class is to implement an application-specific semantics, while an LLVO class is platform-dependent while providing a platform-independent interface to the HLVO class. When implementing a VO class, an HLVO class implementation is derived from an LLVO class implementation, as shown[2] in Figure 1. In the following subsections, we describe main characteristics of the LLVO and HLVO class, and show an application to a heterogeneous system.

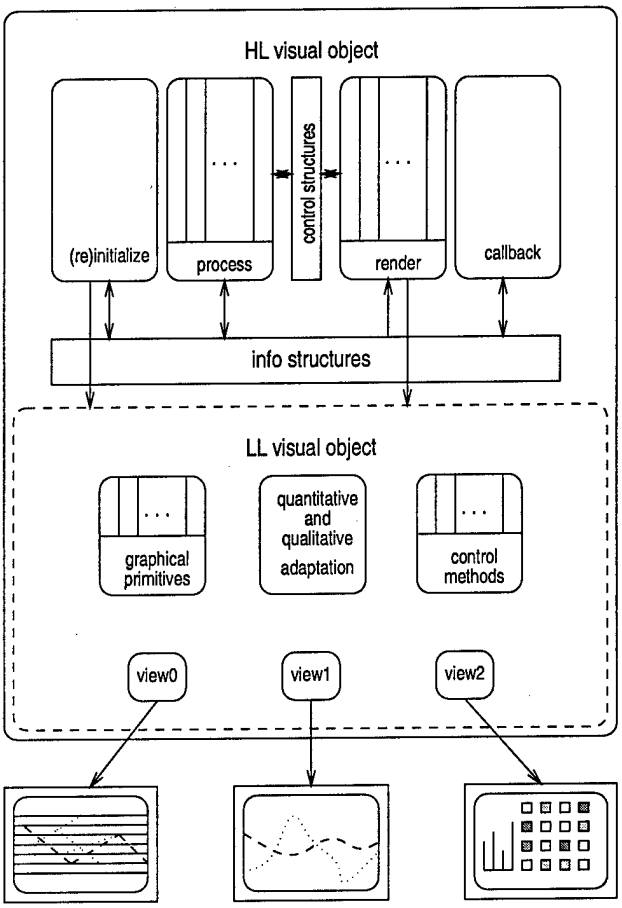


Figure 1. The design of a visual object

2.1 Low-level visual object

The responsibilities of an LLVO class described below illustrate the basic building block of our PAV technology. They have evolved by repeat of substantial experimentation with an X library-based two-dimensional LLVO class that we have implemented in C++.

Multiple views. An LLVO maintains a number of display areas, referred to as *views*. In our implementation of the LLVO class, each display area is supported by a contained object that maintains the state of the corresponding X window.

Graphical primitives. The LLVO class provides methods for rendering simple graphical objects, text and figures in the views. The coordinate system used for the graphical objects' representative coordinates (as arguments to the methods) is a world coordinate system specified by the user at the moment of (re)initializing a view.

Display area. A view consists of an internal area surrounded by *margins*, referred to as *scrollable area*. As a visualization progresses, the mapping from the world coordinate system to the view coordinate system may change, at which point only the contents of the scrollable area may be translated or rescaled (zoomed) as a response.

Control methods. Methods such as *scroll*, *resize*, *rescale* and *snapshot* provide explicit control over each view. Combined with the graphical primitives, they allow an HLVO to control explicitly, among other things, what to be drawn and what to be visible at a *point in time*.

Quantitative adaptation. A relation between a view and calls to graphical primitives that draw in the view may be established that causes the view to adapt dynamically by translating or rescaling (zooming) the contents of the scrollable area, thus implicitly controlling what should be visible over an *interval of time*.

[2]Vertical bars in a high-level method denote the presence of multiple peer components.

Qualitative adaptation. The LLVO class may be portable to multiple graphical platforms that differ at some extent (e.g., different X servers may use different color maps). At run time, it may adapt to the platform capabilities, as well as provide drawing optimization.

An LLVO class implementation may perform bookkeeping about graphical objects being drawn and/or be based on vector graphics, in order to facilitate quantitative and/or qualitative adaptation. However, this is not mandatory and an HLVO class implementation can only assume that the underlying LLVO class is memoryless and raster-based.

The quantitative adaptation of a view in our implementation is initialized by specifying directions (from $\{x+, x-, y+, y-\}$) and types of adaptation (rescaling or scrolling) for these directions. On the other hand, one of the parameters of every graphical primitive is the *adaptation flag* that determines whether the view should adapt before the graphical object is drawn, in order for the graphical object to be visible. In the case of rescaling, the view may also adapt in the opposite way. For example, if the view had to rescale "down" (zoom out) in response to a peak in a temporal line plot[3], it will rescale "up" (zoom in) once the peak has disappeared from the scrollable area. Another parameter for the initialization is the *adaptation quality*. We use it to specify the maximum size of data structures (in our implementation, interconnected red-black trees [6]) used to remember extreme points of graphical objects that have been drawn with the adaptation flag set.

2.2 High-level visual object

Similarly as for the LLVO class in general, we give implementors freedom to define a precise framework for developing HLVOs. In this section, describe our HLVO implementation base, and in Section 3 we present an HLVO development framework. The main components of an HLVO class are the four methods shown in Figure 1.

Event processing. The performance data passed to an HLVO via calls to the processing method are termed *events* (or data events). Based on the events, this method (1) updates performance information referred to as *info structures*, and (2) controls the rendering of this information by updating data structures referred to as *control structures*.

Information rendering. The rendering method may be called, to *map* a portion of the info structures' contents to the LLVO views, either immediately after processing an event (asynchronous rendering mode) or by a thread that may synchronize the rendering of multiple HLVOs (synchronous rendering mode). This method communicates with the processing method by *both reading and writing* the control structures.

Callback processing. An HLVO may also respond to changes in its run-time environment, as well as to the user's commands. This method may, for example, preprocess callback events coming from the LLVO, a GUI, etc., and then forward them to the processing method as if they were data events.

(Re)initialization. In on-line performance visualization, it is desirable to be able to reinitialize partially or reconfigure an HLVO without interrupting the target application and/or instrumentation system that supplies performance data.

In order to allow for rapid prototyping of HLVOs and further PAV research, we have developed a framework based on an implementation of the Scheme language [5] called GUILE [14]. A generic HLVO class inherits the X library-based LLVO class. Both classes have some methods and data wrapped by Scheme procedures within a (run-time) tool integration environment for instrumentation and performance visualization, called PG^{RT}-TIE [1]. The GUI of a VO (in addition to LLVO callbacks) is implemented separately, using a GUILE interface to Tk [25]. It is possible in Scheme, as a dynamically-typed language, for the event processing method to receive any type of data structure as an event, which allows for easy integration of different performance data sources. Most importantly, this high-level algorithmic language is suitable for easy definition of complex info structures (e.g., association lists serving as micro-databases) and compact expression of updating and querying them in the event processing and information rendering methods, respectively. We have developed a CORBA interface for this framework so that a PAV application may consist of VOs distributed over multiple nodes, while a Scheme-to-C compiler [22] is also available that may speed up the HLVO code.

2.3 Application of visual objects to a heterogeneous system

As part of the PG^{RT} environment, two prototype visual objects have been applied to the study of a distributed multimedia real-time application. The target system consisted of a server and a number of heterogeneous receivers of multimedia data streams. The visual objects helped determine the processing demands required to playback different patterns of video frames and to handle different sizes of video frames, as well as the wasted computation due to receiving video frames that cannot be replayed due to time constraints

[3] The contents of the scrollable area is scrolled to the left as the time progresses.

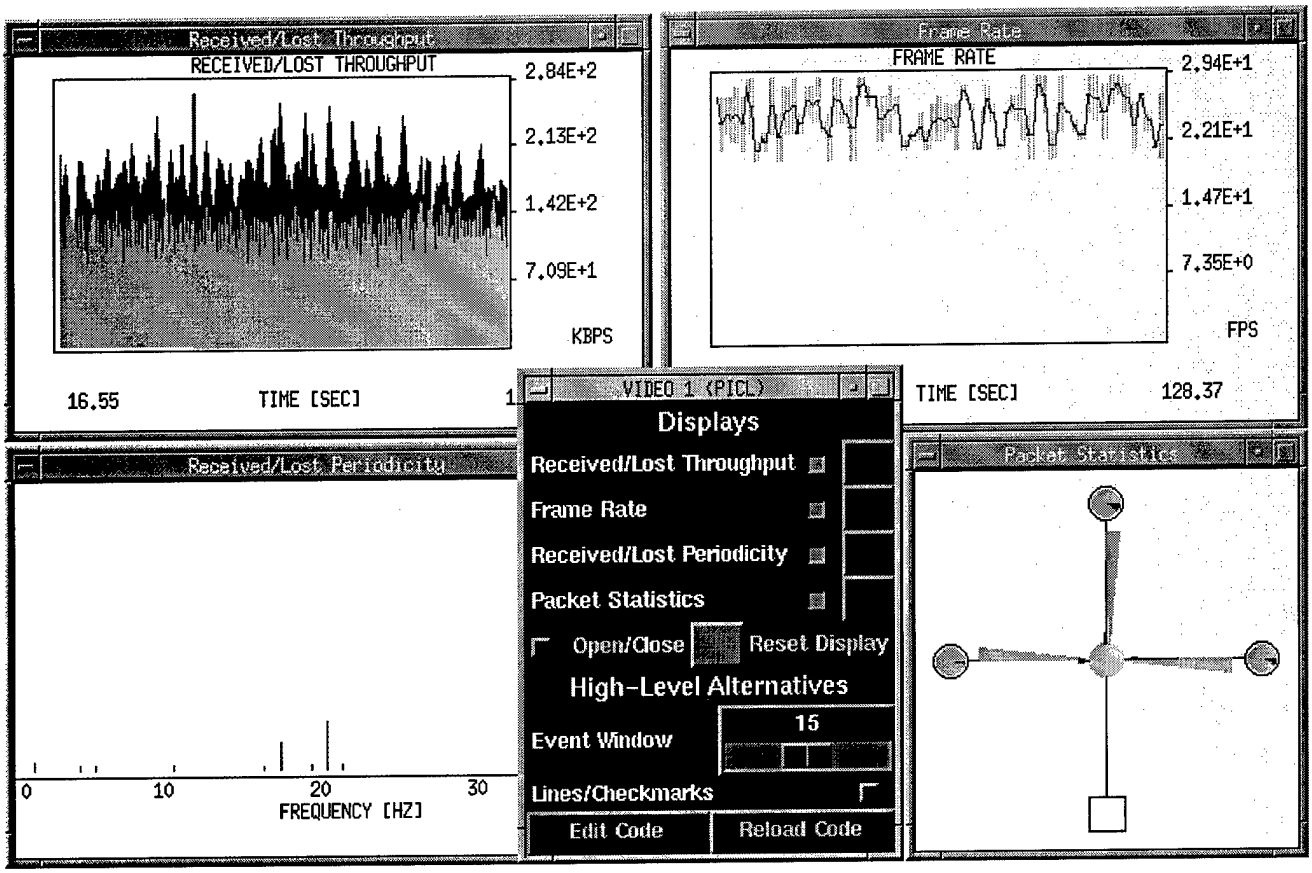


Figure 2. On-line performance visualization of the real-time multimedia application

(e.g., a new video frame arrives before an older video frame can be processed). Furthermore, they helped understand the operation of the application: variations in periodic behavior, and specific points in a network where frame loss occurs, either due to network congestion or individual workstations loading conditions, were displayed. Immense amount of state information, condensed into a set of visual displays, could be used by the feedback control algorithm to make decisions automatically about target bandwidth being requested of the video source.

The snapshot of one visual object is given in Figure 2. Its four views show (1) the throughput of received and estimated throughput of lost video data, (2) the frame rate, (3) the frequency distributions of received and lost video frames over one-second intervals, and (4) a spatial, animated view of all receivers and their connections, depicting the relative volumes of received, lost, used, and dropped video packets. The other visual object has 16 views, divided into four groups: (1) CPU utilization, (2) the periodicity of video frames received, the number of received ATM cells, and (4) the number of lost ATM cells. In each group, there are four related views of the corresponding metric: (1) minimum-average-maximum, (2) sample deviation, (3) aggregate, and (4) per-receiver histogram.

3 Visual Object Markup Language (VOML)

In this section, we describe a framework for semi-automatic design and prototyping of HLVOs. We first define a generic architecture for event processing and performance information rendering that is orthogonal[4] to the VO architecture described in Section 2. Next we present salient characteristics of a very high-level language based on this architecture, called Visual Object Markup Language (VOML), and its compiler that we have designed and implemented. The VOML system allows a performance visualization developer to concentrate on application- and visualization-specific semantics and build HLVOs by combining reusable components.

[4] In this context, where the two architectures coexist, orthogonal means that either architecture can be extended without affecting the other.

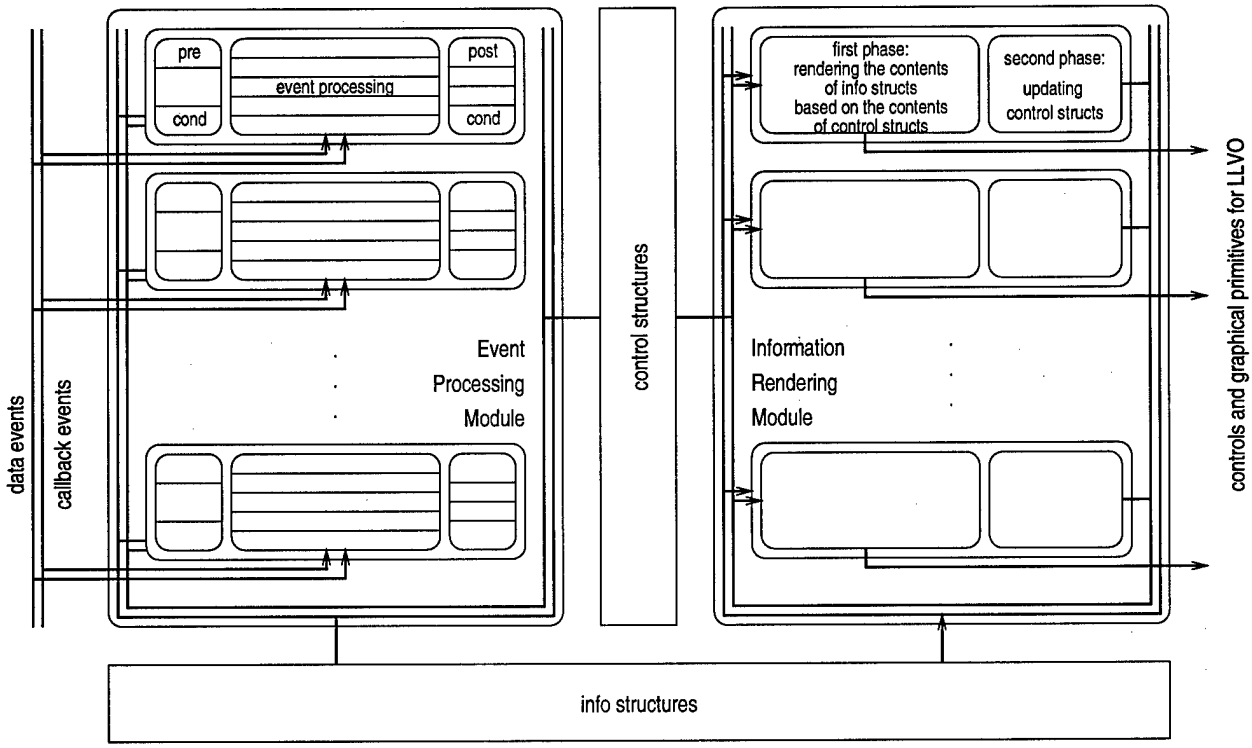


Figure 3. Event Processing and Information Rendering Architecture (EPIRA)

3.1 Event Processing and Information Rendering Architecture (EPIRA)

There are many possible patterns for development of complex HLVOs. For example, one could extend or modify the VO architecture in Figure 1 and build complex HLVOs in a pure object-oriented style, by inheriting from simpler HLVOs. However, since our goal was to develop a framework that could be applicable to target languages that do not have strong support for object orientation (e.g., Scheme and C), we have taken a component-based approach.

Figure 3 shows the *Event Processing and Information Rendering Architecture (EPIRA)*. The architecture specifies the tentative parts of the HLVO architecture, shown in Figure 1, and focuses on the data-driven computation aspect. The two modules in the figure correspond to the event processing and information rendering methods. The events arrive (via method calls) from the two "busses" on the left: they carry the performance data and the changes in the runtime environment. Arrows are drawn to denote unidirectional data flows.

The event processing module may contain a number of *event processing (EP) components*. Each EP component in turn may contain a number of *parts* (separated by horizontal lines in the figure), belonging to one of three classes: (1) event-based ones, shown in the middle and executed upon arrival of a specific event, and condition-based ones, which can be executed (2) before or (3) after the event processing, provided that a specific condition tests true[5]. Since only one event can be received at a time, among all event-based parts (belonging to different EP components) only those for the received event are executed. The conditions corresponding to the condition-based parts are evaluated each time any event is received.

Similarly, the information rendering module contains a number of *information rendering (IR) components*. Each IR component in turn contains two parts, or phases. During the first phase, info and control structures are analyzed and the contents of the info structures is rendered in multiple views appropriately. Once that all first-phase parts of the IR components have been executed, the execution of second-phase parts begins, when the control structures may be updated safely. Since the HLVO assumes that the LLVO has no special rendering support (e.g., a depth buffer), some visualizations may depend on the relative execution order of the first-phase parts.

[5]In VOML, this is referred to as a "condition event."

3.2 The VOML language

We have chosen SGML [9] as the basis for a PAV information infrastructure we plan to build around the VO and EPIRA architectures. For a start, the VOML is an SGML document type definition (DTD) that encompasses the *structure* of HLVOs based on EPIRA. Some of its higher-level elements and example relations among the elements are given in Figures 4a and 4b.

As it can be seen from Figure 4b, VOML attributes are used both to specify certain characteristics of software components described by the elements and to create relations among them, some of which directly correspond to the connections shown in Figure 3. The others are not as "hard-wired," and are described using Figures 4b and 5. Figure 5 defines an example IR component that is used in a visual object defined in Figure 4b.

Although SGML is a very suitable tool for writing structured specifications, it lacks the means for describing semantics of a specification. On the other hand, Scheme is a standardized language with simple syntax and clean semantics that is very suitable for describing the semantics of EP and IR components. Hence, we have decided to imbed Scheme into VOML markup. Combining markup and a programming language, typically Java in WWW-related markup languages, is not a new idea. However, the integration of VOML and Scheme is tighter, as can be seen from the code example in Figure 5. Unlike script-augmented HTML files that are final documents to be "executed," VOML specifications are to be compiled.

Namely, within Scheme code defining the semantics of a component, there may exist references to "formal parameters:" info structures, control structures, events (in EP components) or views (in IR components). The reference notation is `Tn[/m]`, where

- `T` is `$` for info structures, `%` for control structures, and `^` for events and views;
- `n` is the position of the formal parameter in the corresponding parameter list (e.g., `$0` corresponds to `currenttime` in the last line of Figure 4b, because it is the 0-th argument supplied via the `infos` attribute);
- optional `/m` is used for referencing individual fields of an event[6]. For example, an occurrence of `^0/1` within code of EP component `onescalarprocess` in Figure 4b would reference field `value` of data event `onescalar`.

The info and control structures are translated into special global variables by the compiler. Effectively, they are "passed" to EP and IR components by reference when listed in the `infos` and `controls` attributes of the enclosing VOML element. In this way, a reusable component may be written, tested, and placed into a library. In an SGML system, such components may be kept as external SGML entities and used in VOML specifications of different HLVOs by simply referencing them by names.

EP components tend to be application-specific, as they process application-specific event records. To make them more reusable, the element `preprocess-inputs` is provided that allows for specifying "glue logic" (as Scheme expressions) for data events specific to a new application. Namely, before an existing EP component is referenced (i.e., used), any fields of the data events it processes may be arbitrarily preprocessed. For example, an EP component that updates info structures for a simple line-plot visualization (e.g., Scheme code just under `<ep-component name="onescalarprocess"...>` in Figure 4b, updating info structures to be rendered by the IR component code in Figure 5) can be used to visualize the frame rate of a multimedia application. The glue logic in this case could be a function that divides the number of frames received in a time interval[7] by the length of the time interval, whose result would be assigned to the second field (named `value` in the case of the default data event `onescalar`).

Similarly, different library IR components that are parameterized may be combined in interesting ways over a number of views. Additionally, they may be given attributes to determine their higher-level behavior[8]. One such attribute is named `refresh`, which currently can have any combination of values `resize`, `rescale` and `update`. If any of the first two values is used for an IR component, the component will *redraw* its contents if any of the views it draws to get resized or rescaled. This is useful for raster-based LLVO class implementations, where resizing or rescaling an image is lossy. If `update` is used, the component will *undraw* what it drew last time, before proceeding to render the contents of the info structures again. Certain higher-level behavior, which would by default ignore any control structures, can also be controlled by an `enable` attribute that takes a Scheme expression evaluating to a Boolean value. When combining IR components, the HLVO developer may define their execution order.

[6] Currently, VOML only supports PICL [26] compatible events, i.e., lists with the first two elements being integers that determine the record and event type.

[7] Assume that the number of frames is contained in a data event field, and the time interval is kept in an info structure.

[8] Currently, this behavior is supported in our HLVO implementation by auxiliary Scheme code; it might also be supported by an optimizing LLVO class.

```
voml
  head
  body
    visual-object
      event-declarations
        data-event
      info-structures
      control-structures
      utility-code
      view-initializations
        view
      event-processing
        ep-component
          preprocess-inputs
      info-rendition
        ir-component
          line
```

(a) Higher-level elements

```
<event-declarations>
  <data-event name="onescalar" rtype="entry" etype="3000">
    <data-field name="key">
    <data-field name="value">
  ...
<info-structures>
  <variable name="currenttime" type="real">
  <variable name="assoclist" type="list">
  <variable name="palette" type="list">
  ...
<control-structures>
  <variable name="beepcount" type="int" init="0">
  ...
<utility-code>
  (define (beep)
    (display #\Bel))
</utility-code>
<view-initializations>
  <view name="lineplotview" title="Multi-scalar line-plot"...>
...
<event-processing>
  <ep-component name="onescalarprocess" inputs="onescalar.key.value"
    infos="currenttime assoclist">
  ...
<info-rendition>
  <ir-component name="lineplotrender" views="lineplotview"
    infos="currenttime assoclist palette" controls="beepcount">
```

(b) Relations among elements of a VOML specification

Figure 4. A brief description of VOML

```
<description>
  This IR component draws a line-plot of multiple scalars over time, in the supplied
  view (^0). Only lines with the last-update time equal to the current time are drawn.
  Once 10 lines have been drawn, a short sound (beep) is generated.

  The info structures consist of the current time ($0, non-negative real number)
  a multi-scalar association list ($1, indexed by non-negative integer keys),
  and a color palette ($2, a list of strings -- color names).
  Each value in the association list is a 4-element vector:
  #(old-time old-value new-time new-value).

  The key of each value in the multi-scalar association list is used to index the
  color. When all colors are exhausted, the line thickness is increased to distinguish
  between different scalars. A counter is used as a control structure (%0) for
  generating sounds.
</description>
  (let ((palette-len (length $2)))
    (alist-for-each
      (lambda (scalar-id scalar)
        (if (= $0 (vector-ref scalar 2))
          (begin
            (set! %0 (+ %0 1))
            (if (= %0 10)
              (begin
                (beep)
                (set! %0 0)))
            <line view="^0" from="(vector-ref scalar 0) (vector-ref scalar 1)"
              to="$0 (vector-ref scalar 3)" thick="(+ (quotient scalar-id palette-len) 1)"
              color="(nth (modulo scalar-id palette-len) $2)" adapt="yes" clip="margin">)))
      $1))
```

Figure 5. Code of the IR component used as `lineplotrender` in Figure 4b

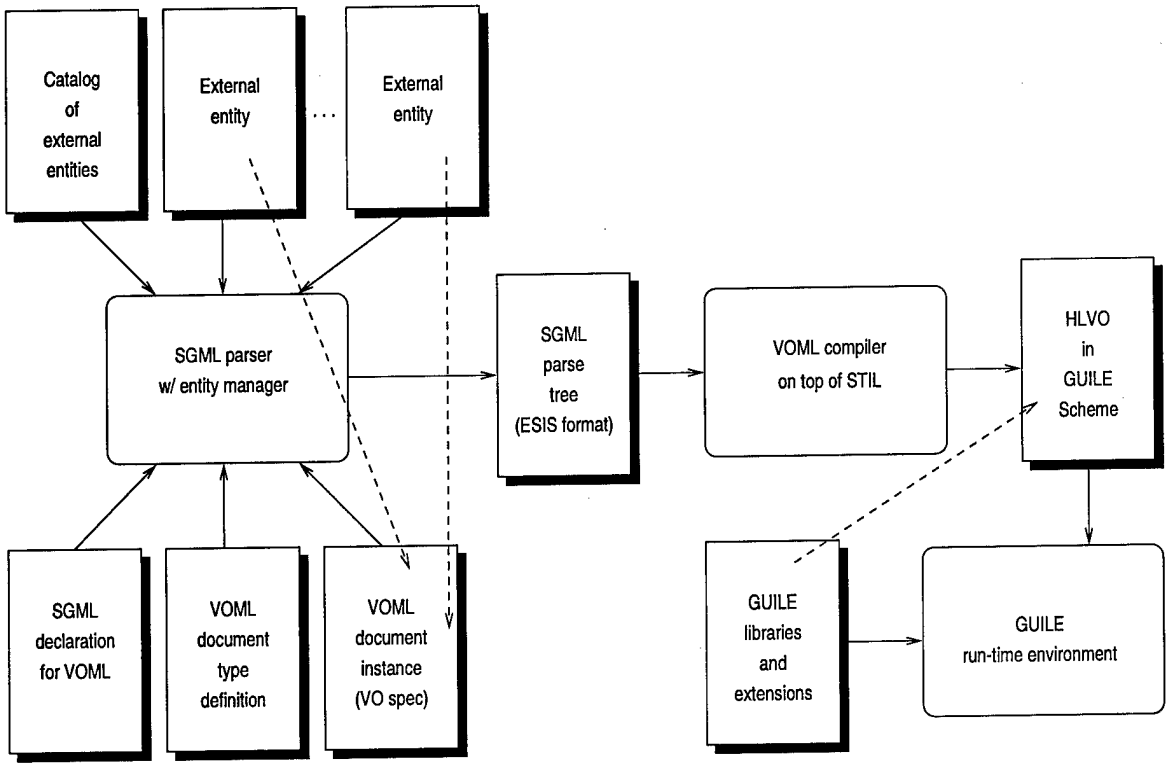


Figure 6. VOML compilation and execution process diagram

3.3 The VOML compiler

The VOML compiler is built on top of an SGML transformation library called STIL [21] and consists of the following components.

SGML parser. The `sgmls` parser [4] is used as the front-end that parses an SGML declaration, VOML DTD, and external entities used in a VOML specification of an HLVO.

STIL library. This library is written for the `clisp` [10] implementation of Common Lisp with CLOS. It allows traversing a parse tree created by the SGML parser, and defining "hooks" (semantic actions) that are called during the traversal.

VOML validating parser. One part of this component consists of the hooks called by the STIL library. The other part consists of CLOS objects that contain code and other information relevant to EPIRA components of the HLVO specification being compiled. The hooks process VOML elements (including the contents in Scheme), their relations and attributes, and build the CLOS objects.

VOML code generator. This component "tangles" the plain, application- and visualization-specific Scheme code from a VOML specification with code that it generates for integration with the run-time environment. The latter includes a graphical user interface for accessing and modifying selected info and control structures, managing views, registering VOs with routines that supply data events, etc.

Figure 6 shows the compilation and execution process of a VOML specification in our GUILE-based environment. Processes are shown as oval rectangles, while input, intermediate and output files are shown as rectangles. Solid lines denote the process of file inclusion, while dashed lines denote references in VOML and Scheme files.

The VOML design allows for extending the compiler to support other Scheme-based run-time environments. Interesting extensions would be for Kawa [2], a Scheme compiler written in Java that generates JVM bytecodes, and Skij [13], a Scheme interpreter that allows rapid prototyping in the Java environment. With a LLVO class implementation in Java, it would become relatively easy to develop PAV applications running in a WWW browser, leveraging platform-independent VOML specifications, and receiving performance data over the Internet. Alternatively, a simpler version of VOML could be defined in XML [15] instead of

SGML, and/or Java could be used instead of Scheme to both compile and execute VOML specifications.

While an SGML parser uses an *entity manager* to find components of a document which are referenced as external entities—such as library EP and IR components—within its virtual storage system, the SGML standard itself does not specify how to implement one. A WWW-enabled entity manager would further enlarge the PAV information infrastructure and automatize monitoring and PAV of globally distributed applications. Figure 7 shows an example of how component definitions could be fetched from a WWW site by the entity manager, to be included for compilation. In the example, the vendor of an imaginary software product, whose performance we want to visualize, keeps the latest implementations of an EP and IR component for the product, ready to be used in our HLVO specification[9].

```
<!DOCTYPE VOML PUBLIC "-//MSU-PGRT//DTD VOML 1.0//EN"
  [
     <!ENTITY SoftwareXYZep
       SYSTEM "http://vendor.com/voml/XYZep.voml">
     <!ENTITY SoftwareXYZir
       SYSTEM "http://vendor.com/voml/XYZir.voml">
  ]
>
<voml>
...
      <event-processing>
        <ep-component name="XYZep"
                      inputs="mydata.f1.f2" ...>
          &SoftwareXYZep;
        </ep-component>
...
      <info-rendition>
        <ir-component name="XYZir"
                      views="myview" ...>
          &SoftwareXYZir;
        </ir-component>
...
```

Figure 7. Sketch of a VOML specification that uses remote component definitions

4 The VOML Specification of a Simple Visual Object

In this section, we present and comment on main parts of the VOML specification of a simple VO with a view similar to the last one of the VO shown in Figure 2. The VO receives performance data events from a distributed application, generated whenever a node is (1) added or (2) removed, and periodically to carry profile data from each node. The event declaration section is shown in Figure 8. The record and event types of the first two events are taken from the PICL specification [26], while the profile event belongs to an extension of PICL. When some field are skipped (i.e., ignored), the `index` attributed is used to specify the position of the next declared field.

```
<event-declarations>
  <data-event name="addnode"
              rtype="pg-entry" etype="-901">
    <data-field name="ts" type="int">
    <data-field name="node-id" type="int">
  </data-event>
  <data-event name="rmnode"
              rtype="pg-exit" etype="-901">
    <data-field name="ts" type="real">
    <data-field name="node-id" type="int">
  </data-event>
  <data-event name="node-prf"
              rtype="entry" etype="3141">
    <data-field name="ts" type="real">
    <data-field name="node-id" index="3" type="int">
    <data-field name="node-type" index="5" type="int">
    <data-field name="rkbps" type="real">
    <data-field name="tb" type="real">
    <data-field name="used" type="real">
    <data-field name="fps" type="real">
    <data-field name="packets" type="int">
    <data-field name="pack-used" type="int">
  </data-event>
</event-declarations>
```

Figure 8. Event declarations

The info and control structure specifications are shown in Figure 9. The info variable `numofnodes` (although redundant) keeps the current number of communicating nodes; `nodes` is an association list that keeps the previous and current profile of each node; `nodeno` keeps the (non-negative) node id from the latest profile event. The control variable `nodechange` indicates whether the number of nodes has changed (meaning that the view has to be updated, as will be seen later).

```
<info-structures>
  <variable name="numofnodes" type="int">
  <variable name="nodes" type="list">
  <variable name="nodeno" type="int" init="-1">
</info-structures>

<control-structures>
  <variable name="nodechange" type="boolean">
</control-structures>
```

Figure 9. Info and control structures

The next is the utility code section, which we omit for brevity. It contains function `getfontname` that returns a font name from a list of available fonts, given some hints. Besides, functions `id-get`, `id-put` and `id-rem`, which are used to manipulate the `nodes` association list, may be defined in this section (in our case, they are defined and exported from another module, available in the run-time environment).

The view initialization section is shown in Figure 10. The (only) `BU-View` view is 700 by 700 pixels, through which a rectangle in the world coordinate system from

[9] It is assumed that we already have the information about the components' interfaces.

$(-10,-10)$ to $(110,120)$ is visible. In this example, the view neither scrolls nor zooms. The control variable nodechange is set to true only to trigger the drawing of the switch in the beginning.

```
<view-initializations>
  <view name="BU-View" window="700 700"
    world="-10 110 -10 120" controls="nodechange">
    <description>Switch, nodes and bandwidth
      utilizations</description>
    (set! %0 #t)
  </view>
</view-initializations>
```

Figure 10. View initialization

```
<event-processing>
  <ep-component name="rmnode" inputs="rmnode.ts.node-id"
    infos="numofnodes nodes nodeno" controls="nodechange">
    <description>Remove a node</description>
    <input name="^0">
      (set! $1 (id-rem $1 ^0/1))
      (set! $0 (- $0 1))
      (set! $2 -1)
      (set! %0 #t)
    </input>
  </ep-component>
  <ep-component name="addnode" inputs="addnode.ts.node-id"
    infos="numofnodes nodes nodeno" controls="nodechange">
    <description>Add a node, reset the infos</description>
    <input name="^0">
     (set! $1 (id-put $1 ^0/1
           (cons (vector ^0/0 0 0 0 0 0 0 0 0 0)
                 (vector ^0/0 0 0 0 0 0 0 0 0 0))))
     (set! $0 (+ $0 1))
     (set! $2 -1)
     (set! %0 #t)
    </input>
  </ep-component>
  <ep-component name="nodeprofile"
    inputs="node-prf.ts.node-id.node-type.rkbps.tb.used.fp
s.packets.pack-used"
    infos="numofnodes nodes nodeno">
    <description>Update a node's infos</description>
    <input name="^0">
      (let ((old-info (cdr (id-get $1 ^0/1)))
            (new-info (vector ^0/0 ^0/1 ^0/2 ^0/3 ^0/4
                              ^0/5 ^0/6 ^0/7 ^0/8)))
        (set! $1 (id-put $1 ^0/1
                         (cons old-info new-info))))
      (set! $2 ^0/1)
    </input>
  </ep-component>
</event-processing>
```

Figure 11. Event processing components

There is one EP component for each event, although one could implement, for example, only one for all the three events. They are shown in Figure 11, as updating the info and control structures according to the event declarations. Fields of an event are listed using a notation in which the event name is followed by some of its fields' names, delimited by periods. In the nodeprofile EP component, all the event fields declared above are used. It is not necessary to use them all and in the same order as declared; the ^m/n

```
<ir-component name="nodes-ir" views="BU-View"
  infos="numofnodes" controls="nodechange"
  refresh="update resize" buffer="yes" enable="%0">
  <description>Switch, nodes, connections</description>
  (let* ((viewinfo <view-info view="^0">)
         (width    (list-ref viewinfo 5))
         (height   (list-ref viewinfo 6))
         (size     (inexact->exact
                     (max (/ width 40) (/ height 40))))
         (font     (getfontname "fonttable"
                                "courier" size "bold")))
    <text view="^0" coords="50 107" halign="CENTER"
      font="font" fcolor='"black"'
      content='"Bandwidth Utilization"'>
    <figure view="^0" filename='"bggif/switch.gif"'
      orig-origin="0 0" orig-extents="0 0"
      world-origin="45 45" world-extents="10 10">
    (let* ((nodenum (- $0 1))
           (step    (/ 6.28 nodenum)))
      (if (gt nodenum 0)
        (let loop ((num nodenum))
          (let* ((angle  (* num step))
                 (sine   (sin angle))
                 (cosine (cos angle)))
            <figure view="^0"
              filename='"bggif/node.gif"'
              orig-origin="0 0" orig-extents="0 0"
              world-origin="(+ 45 (* 45 sine))
                            (+ 45 (* 45 cosine))"
              world-extents="10 10">
            <line view="^0" from="(+ 50 (* 6 sine))
                                  (+ 50 (* 6 cosine))"
                  to="(+ 50 (* 39 sine))
                      (+ 50 (* 39 cosine))"
              color='"red"' thick="12">
            (if (gt num 1)
              (loop (- num 1))))))))
  <end-with>(set! %0 #f)</end-with>
</ir-component>
```

Figure 12. Template IR component

notation uses the order(s) given in the inputs attribute. It can be seen that the field node-id is used as the key, and the value field in the nodes association list is a pair of vectors keeping the previous and current profile of a node. In this example, only the current profile will be used, but in a more complex VO both the previous and current one may be needed.

Finally, the information rendering section consists of two IR components. The nodes-ir IR component, which is shown in Figure 12 and will be executed first[10], writes text and draws the switch and as many "PG^{RT} globes" around it as there are active nodes, connected with the switch via thick red lines. The enable attribute specifies that this IR component should be executed whenever the number of nodes has changed. The refresh attribute adds that everything the IR component drew last time should be redrawn when the view BU-View is resized. It also specifies that everything the IR component drew last time has to be undrawn before something new is drawn (whenever the IR component is enabled). The buffer attribute is used

[10]In the prototype implementation of the VOML compiler, the execution order of the IR components is opposite of the order they appear in a specification.

```
<ir-component name="bu-ir" views="BU-View"
  infos="numofnodes nodes nodeno" refresh="resize">
  <description> Bandwidth utilization </description>
  (if (gt $2 -1)
    (let* ((angle    (* $2 (/ 6.28 (- $0 1))))
           (sine     (sin angle))
           (cosine   (cos angle))
           (new-info (cdr (id-get $1 $2)))
           (newkbps  (vector-ref new-info 3))
           (newused  (vector-ref new-info 5))
           (mag      (* 25 (/ newused newkbps))))
      <line view="^0" from="(+ 50 (* 6 sine))
                            (+ 50 (* 6 cosine))"
                      to="(+ 50 (* 39 sine))
                          (+ 50 (* 39 cosine))"
        color='"red"' thick="12">
      <line view="^0" from="(+ 50 (* 6 sine))
                            (+ 50 (* 6 cosine))"
                      to="(+ 50 (* (+ 6 mag) sine))
                          (+ 50 (* (+ 6 mag) cosine))"
        color='"blue"' thick="10">
    (set! $2 -1)))
</ir-component>
```

Figure 13. Active IR component

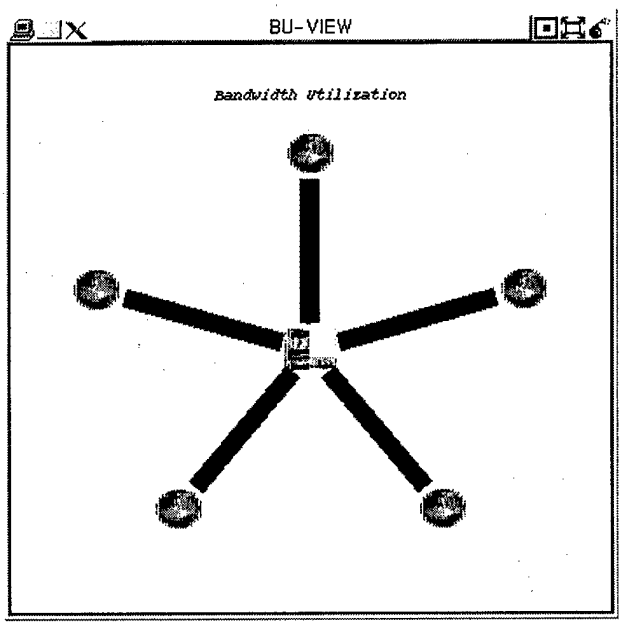


Figure 14. A snapshot of the view

to make the IR component draw in-memory only until it is done, and then flush the contents of the memory to the screen. This is useful to make the rendering smoother and faster when there are many graphical objects to be drawn. This IR component resets the `nodechange` control variable in the second phase, so that other IR components may be added safely that depend on the value of this variable[11].

The other IR component is executed each time an event is received, and it draws a blue thick line on top of a red thick line (drawn in the same place as the thick red line drawn by the former IR component, so that it can effectively be undrawn), showing the relative bandwidth used by a node (if a profile event was received last that set the `nodeno` info variable to a non-negative value). Its code is shown in Figure 13. The refresh attribute treats the resizing of the view same as above.

A snapshot of the view is given in Figure 14. We have not shown all the VOML features in this example; for a tutorial, please visit the URL given in Section 6.

5 Related Work

ParaGraph [11] is a PAV tool for parallel programs, based on the PICL communication library. PAV environments are progressing with features to incorporate new analysis and display modules. Visualization environments are not only becoming extensible, but retargetable to different analysis scenarios. Pablo took this research one step further by incorporating support for performance environment prototyping [20]. VIZ continues in this direction by focusing on the visualization technology required for application-specific performance visualizations [12]. Avatar [19] uses Pablo to study two types of high-performance input/output of the Portable Parallel File System (PPFS): parallel scientific codes and WWW servers. The Rivet project [17] integrates new visualization tools into the design and evaluation process of a variety of computer system components, specifically processor and memory systems, multiprocessor architectures, compilers, operating systems, and networks. Lucent Technologies' Visual Insights [24] offers a set of interactive and linked data visualization components for the Microsoft ActiveX developer market that help software developers to create more flexible, animated ways to display trends in vast stores of information.

In Table 1 we compare PG^{RT} visual objects with related PAV tools and systems. While some of the latter have gone farther in certain direction, such as the graphical metaphor, our design decisions were primarily based on the requirements stated in Section 1. We have also been concerned that insisting on state-of-the-art, academic software technologies, such as, for example, lazy functional languages, could limit the practicality of our approach. Instead, using technologies that are gaining acceptance, such as SGML (e.g., in the Chemical Markup Language [18]) and use of structure, components and scripting (e.g., in VRML [23]), we hope to contribute to the PAV community.

[11] In Scheme, this second phase is implemented using `delay` and `force`.

Tool/System	On/off-line operation	Graphical metaphor	Underlying graphical technology	View classes	Reusable
ParaGraph	off-line	data-flow	X library	generic	no
Pablo widgets	off-line	data-flow	X library	generic	yes
Avatar	on-line	data-flow	VRML	scattercube only	no
VIZ	on-line	data-reactive	Open Inventor	domain-specific	yes
Rivet	off-line	data-flow	OpenGL	domain-specific	yes
Visual Insights	off-line	n/a	n/a	generic	yes
PG^{RT} VO	on-line	data-flow	low-level VO implementations	domain-specific	yes

Table 1. Performance visualization tools and systems

6 Conclusions

We have presented a novel PAV technology intended to satisfy growing needs by researchers and users of parallel and distributed systems. Salient characteristics of the technology include support for rapid prototyping and automated design of PAV tools, object orientation, distributability, portability, code reuse and flexibility. Tutorials with examples and reference manuals for PG^{RT}-TIE and VOML can be found in the Documents section at `http://www.egr.msu.edu/Pgrt/`.

In the future, we plan to extend and improve the technology, and make it available to the PAV community. We will develop visual objects specific to parallel/distributed real-time applications, but also try to help PAV developers using our technology in other areas by developing domain-specific libraries of EP and IR components.

References

[1] A. Bakić, M. W. Mutka, and D. T. Rover. Real-time system performance visualization and analysis using distributed visual objects. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, December 2 1997.

[2] P. Bothner. Kawa - compiling dynamic languages to Java VM. In *Proceedings of the 1998 USENIX Annual Technical Conference, New Orleans*, June 19 1998.

[3] D. Box. *Essential COM*. Addison-Wesley, January 1998.

[4] J. Clark. `sgmls`: A validating SGML parser. WWW, 1993. `ftp://ftp.jclark.com/pub/sgmls/`.

[5] W. Clinger and J. Rees, editors. *Revised(4) Report on the Algorithmic Language Scheme*. IEEE, November 2 1991.

[6] Corman, Lieserson, and Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[7] J. Dongarra et al. MPI: A message-passing interface standard. Technical report, Oak Ridge National Laboratory, June 1995.

[8] G. Giest et al. *PVM 3.0 User's Guide and Reference Manual*. ORNL/TM–12187, February 1993.

[9] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[10] B. Haible. CLISP, a Common LISP implementation, 1997. `http://clisp.cons.org/ haible/clisp.html`.

[11] M. T. Heath and J. E. Finger. Visualizing performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[12] H. H. Hersey, S. T. Hackstadt, L. T. Hansen, and A. D. Malony. Viz: A visualization programming system. Technical Report CIS-TR-96–05, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403–1202, April 1996.

[13] IBM alphaWorks. An interactive scripting language for rapid prototyping in the Java environment. WWW, 1998. `http://www.alphaWorks.ibm.com/formula/-Skij/`.

[14] T. Lord. An anatomy of Guile/the interface to Tcl/Tk. *Usenix Tcl/Tk Workshop '95*, 1995. More information available on the WWW, at `http://www.redbean.com/guile/`.

[15] E. Maler. XML specification DTD. WWW, 1998. `http://www.sil.org/sgml/xmlspec--19980323-dtd.txt`.

[16] T. J. Mowbray and R. Zahavi. *The Essential CORBA—System Integration Using Distributed Objects*. The Object Management Group ISBN 0–471–10611–9, 1997.

[17] T. Munzner. Laying out large directed graphs in 3D hyperbolic space. In *Proceedings of the InfoVis '97*, 1997.

[18] P. Murray-Rust. Chemical markup language. Technical report, Venus, `http://www.venus.co.uk/omf/cml/`, 1997.

[19] D. Reed, K. Shields, W. Scullin, L. Tavera, and C. Elford. Virtual reality and parallel systems performance analysis. *IEEE Computer*, 28(11):55–67, November 1995.

[20] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. An overview of the Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 7 1992.

[21] J. Schrod. SGML transformations in LISP. Technical report, Computer Science Department, Technical University of Darmstadt, Kranichweg 1, D-63322 Roedermark, FR Germany, 1995.

[22] T. Tammet. Hobbit: A Scheme-to-C compiler. Technical report, Department of Computing Science, Chalmers University of Technology, University of Göteborg, Sweden, 1995. Available at `http://www-swiss.ai.mit.edu/~jaffer/Hobbit.html`.

[23] The VRML Consortium Inc. The virtual reality modeling language. WWW, 1997. `http://www.vrml.org/`.

[24] Visual Insights, Lucent Technologies. Software components. WWW, 1998. `http://www.visualinsights.com/components/`.

[25] B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall ISBN 0–13–182007–9, 1995.

[26] P. H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831-6367, September 1992.

Biographies

Aleksandar M. Bakić is a research assistant in the Department of Computer Science and Engineering at Michigan State University. His research interests include engineering of complex, distributed real-time systems, integrated instrumentation and performance visualization for parallel and distributed systems, and compiler-based technologies. He received his BS in computer engineering from the School of Electrical Engineering, Belgrade, Yugoslavia, and his MS in computer science from Michigan State University. He is a student member of ACM. Contact him at the Dept. of Computer Science and Eng., 3115 Eng. Bldg., Michigan State Univ., East Lansing, MI 48824-1226; `bakicale@cse.msu.edu`.

Matt W. Mutka is an associate professor in the Department of Computer Science and Engineering at Michigan State University. His research interests include resource-management issues in distributed and parallel systems, real-time systems, high-speed computer networks, and instrumentation and visualization for the design and evaluation of embedded computing systems. He received his BS in electrical engineering from the University of Missouri-Rolla, his MS in electrical engineering from Stanford University, and his PhD in computer science from the University of Wisconsin-Madison. He is a member of the IEEE Computer Society, IEEE Communication Society, and ACM. Contact him at the Dept. of Computer Science and Eng., 3115 Eng. Bldg., Michigan State Univ., East Lansing, MI 48824-1226; `mutka@cse.msu.edu`.

Diane T. Rover is an associate professor in the Department of Electrical and Computer Engineering and the director of the Computer Engineering Program at Michigan State University. Her research interests include integrated program development and performance environments for parallel and distributed systems, instrumentation systems, performance visualization, embedded real-time systems, and reconfigurable hardware. She received her BS in computer science and her MS and PhD in computer engineering, all from Iowa State University. She is a member of the IEEE Computer Society, ACM, and American Society for Engineering Education. Contact her at the Dept. of Electrical and Computer Eng., 2120 Eng. Bldg., Michigan State Univ., East Lansing, MI 48824-1226; `rover@egr.msu.edu`.

Heterogeneous Distributed Virtual Machines in the Harness Metacomputing Framework

Mauro Migliardi and Vaidy Sunderam
Emory University, Dept. of Math & Computer Science
1784 N. Decatur Rd. #100
Atlanta, GA, 30322, USA
{om, vss}@mathcs.emory.edu

Abstract

Harness is a Java-centric, experimental metacomputing framework based upon the principle of dynamic enrollment and reconfiguration of heterogeneous computational resources into distributed virtual machines. The dynamic behavior of the system is not limited to the number and types of computers and networks that comprise the virtual machine, but also extends to the capabilities of the virtual machine itself. These fundamental characteristics address the inflexibility of current metacomputing frameworks as well as their incapability to easily incorporate new, heterogeneous technologies and architectures and avoid rapid obsolescence. The adaptable behavior of Harness derives both from a user controlled, distributed "plug-in" mechanism and from an event driven, dynamic management of the distributed virtual machine status that are central features of the system.

1 Introduction

Harness is an experimental, Java-centric metacomputing framework based upon the principle of dynamic enrollment and reconfiguration of heterogeneous computational resources into networked virtual machines. The reconfiguration capabilities of Harness are not limited to the set of computers and networks enrolled in the virtual machine, but, on the contrary, they also include the services offered by the virtual machine itself. Further on, reconfiguration is not constrained to take place during the virtual machine setup phase but can also be applied at run-time. This level of reconfigurability is allowed by a user-controlled, distributed "plug-in" mechanism together with a dynamic, fault tolerant management of the distributed virtual machine status.

The motivation for a plug-in-based approach to reconfigurable virtual machines derives from two observations. First, new advances in information technology require distributed and cluster computing to adapt to heterogeneous processors, interconnection network types and protocols in order to be able to take advantage of them. For example, the availability of Myrinet [1] interfaces and Illinois Fast Messages [2] has recently led to new models for closely coupled Network Of Workstations computing systems. Similarly, multicast protocols and better algorithms for video and audio codecs have led to a number of projects that focus on tele-presence over distributed systems. In these instances metacomputing frameworks not able to cope with computational resources that are heterogeneous both in architecture and in connectivity are subject to rapid obsolescence. In fact, the underlying middleware either needs to be changed or re-constructed, thereby increasing the effort level involved and hampering interoperability. On the contrary, a virtual machine model intrinsically incorporating reconfiguration capabilities in terms of the services provided by any single computational resource will allow the definition of a consistent service baseline in such an evolving environment and will address these issues in an effective manner.

Second, applications characterized by long run times require a virtual machine environment that can dynamically adapt the available resources to meet the application's needs, rather than forcing the application to fit into a fixed environment. As an example we can cite long-lived simulations such as climate simulations. These applications evolve through several phases, data input, problem setup, calculation, and analysis or visualization of results, each one with its own profile in terms of resource requests and requirements. In order to maximize the availability of resources to the application a metacomputer needs to be able to dynamically enroll heterogeneous computational resources into the resource pool. At the same time, if the metacomputer is not able to reconfigure the resources to suit the needs of the application the availability is purely nominal and the utilization of the

0-7695-0107-9/99 $10.00 © 1999 IEEE

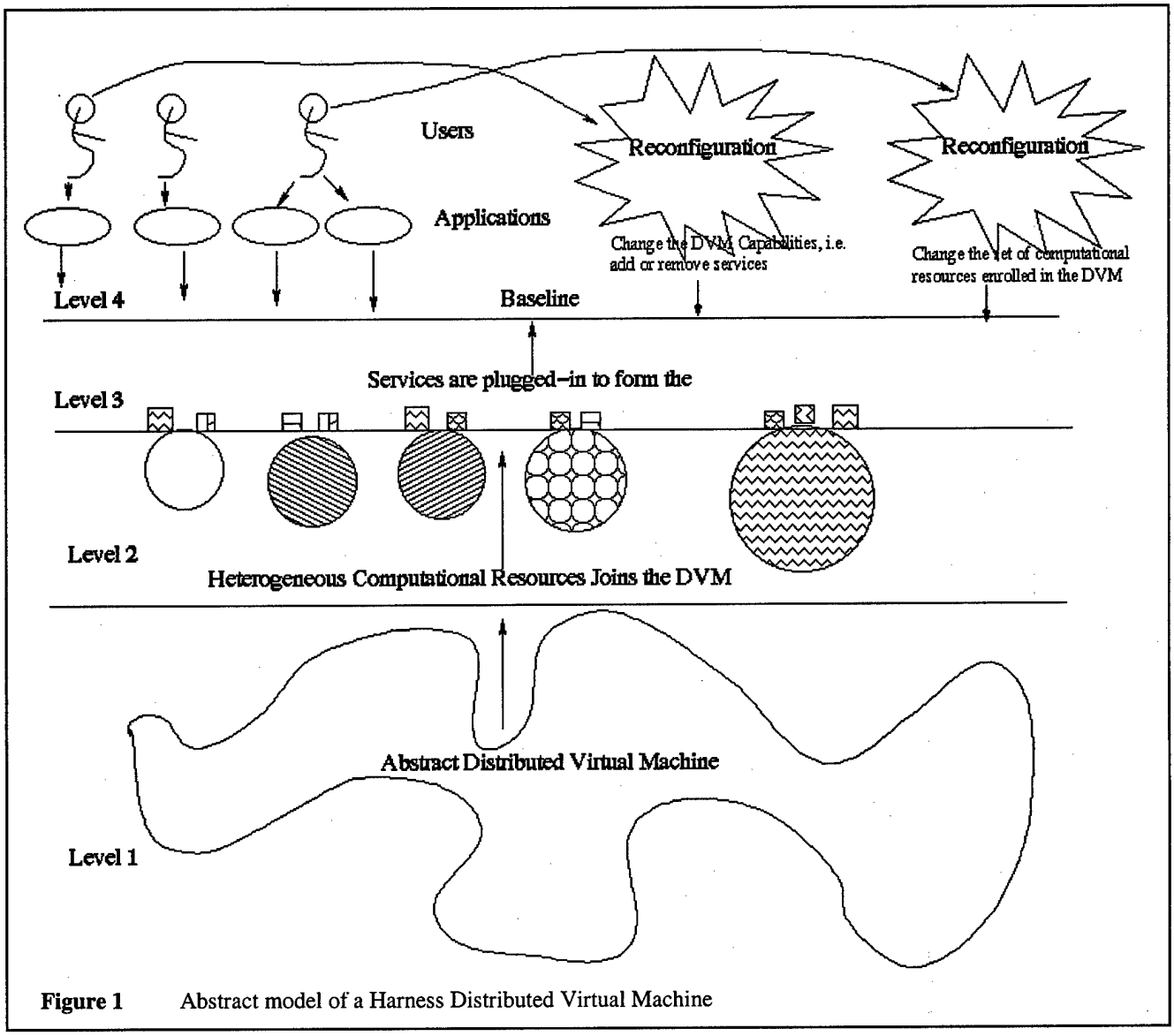


Figure 1 Abstract model of a Harness Distributed Virtual Machine

available resources is poor. On the contrary, by allowing applications to dynamically enroll and reconfigure heterogeneous computational resources at run time, the overall utilization of the computing infrastructure can be enhanced and the need for a consistent service baseline fulfilled.

The overall goal of the Harness project encompasses several different issues such as dynamic reconfiguration and management of distributed virtual machines, fault-tolerance, security, authentication and access control. However, in this paper we focus our attention in investigating and developing a mechanism for fault tolerant, dynamic reconfiguration and management of distributed virtual machines. Within the framework of an heterogeneous computing environment, this mechanism allows users and applications to dynamically customize, adapt, and extend the distributed computing environment's features to match their needs without compromising the consistency of the programming environment itself.

The paper is structured as follows: in section 2 we describe the abstract architecture of Harness DVMs and our design choices; in section 3 we detail the architecture of the system; in section 4 we describe some example plug-ins and applications; in section 5 we relate our work with other similar projects; finally in section 6 we provide some concluding remarks.

2 Fundamental abstractions, terminology and design choices

The fundamental abstraction in the Harness metacomputing framework is the **Distributed Virtual Machine** (DVM) (see figure 1, level 1). Any DVM is associated with a symbolic name that is unique in the Harness name space, but has no physical entities connected to it. **Heterogeneous Computational Resources** may enroll into a DVM (see figure 1, level 2) at any time, however at this level the DVM is not ready yet to accept requests from users. To get ready to interact with users and applications the heterogeneous computational resources enrolled in a DVM need to load **plug-ins** (see figure 1, level 3). A plug-in is a software component implementing a specific **service**. By loading plug-ins a DVM can build a consistent **service baseline** (see figure 1, level 4). Users may **reconfigure** the DVM at any time (see figure 1, level 4) both in terms of computational resources enrolled by having them **join** or **leave** the DVM and in terms of services available by **loading** and **unloading** plug-ins.

The main goal of the Harness metacomputing framework is to achieve the capability to enroll heterogeneous computational resources into a DVM and make them capable of delivering a consistent service baseline to users. This goal require the programs building up the framework to be as portable as possible over an as large as possible selection of systems. The availability of services to heterogeneous computational resources derives from two different properties of the framework: the portability of plug-ins and the presence of multiple searchable plug-in repositories. Harness implements these properties mainly leveraging two different features of Java technology. These features are the capability to layer a homogeneous architecture such as the Java Virtual Machine (JVM) [3] over a large set of heterogeneous computational resources, and the capability to customize the mechanism adopted to load and link new objects and libraries. However, the adoption of the Java language as the development platform for the Harness metacomputing framework has given us several other advantages:

- it allowed us to develop the framework as a collection of cooperating objects with consistent boundaries (Java Classes) and to guarantee to users an OO development environment;
- it allowed us to define a clear and consistent boundary for plug-ins, in fact each plug-in is required to appear to the system as a Java class;
- it allowed us to implement all the entities in the framework adopting a robust multithreaded architecture;
- it allows users to develop additional services both in a passive, library-like flavor and in an active thread-enabled flavor;
- it provided us an Object Oriented mechanism to require services from remote computational resources (Java Remote Method Invocation [4]);
- it provided us a generic methodology to transfer data over the network in a consistent format (Java Object Serialization [5]);
- it allowed us to provide to users the definition of interfaces to be implemented by plug-ins implementing the basic services;
- it allowed us to tune the trade-off between portability and efficiency for the different components of the framework.

This last capability is extremely important, in fact, although portability at large is needed in all the components of the framework, it is possible to distinguish three different categories among the components that requires different level of portability. The first category is represented by the components implementing the capability to manage the DVM status and load and unload services. We call these components **kernel level services**. These services require the highest achievable degree of portability, as a matter of fact they are necessary to enroll a computational resource into a DVM. The second category is represented by very commonly used services (e.g. a general, network independent, message passing service or a generic event notification mechanism). We call these services **basic services**. Basic services should be generally available, but it is conceivable for some computational resources based on specialized architecture to lack them. The last category is represented by highly architecture specific services. These services include all those services that are inherently dependent on the specific characteristics of a computational resource (e.g. a low-level image processing service exploiting a SIMD co-processor, a message passing service exploiting a specific network interface or any service that need architecture dependent optimization). We call these services **specialized services**. For this last category portability is a goal to strive for, but it is acceptable that they will be available only on small subsets of the available computational resources. These different degrees of required portability and efficiency over heterogeneous computational resources can optimally leverage the capability to link together Java byte code and system dependent native code enabled by the Java Native Interface (JNI) [6]. The JNI allows to develop the parts of the framework that are most critical to efficient application execution in ANSI C language and to introduce into them the desired level of architecture dependent optimization at the cost of increased development effort.

The use of native code requires a different implementation of a service for each type of heterogeneous computational resource that need to deliver that service. This fact implies a development effort

multiplied for each plug-in including native code. However, if a version of the plug-in for a specific architecture is available, the Harness metacomputing framework is able to fetch and load it in a user transparent fashion, thus users are screened from the necessity to control the set of architectures their application is currently running on. To achieve this result Harness leverages the capability of the JVM to let users redefine the mechanism used to retrieve and load both Java classes bytecode and native shared libraries. In fact, each DVM in the framework is able to search a set of plug-ins repositories for the desired library. This set of repositories is dynamically reconfigurable at run-time, users can add new repositories at any time.

3 Harness requirements and architecture

3.1 Prototype implementation constraints and long term choices

The main requirement for a computational resource to be enrolled in a DVM is the capability to run Java programs, i.e. the presence of an implementation of a JVM on the given architecture. This is not a constraint derived from prototype implementation but rather a general design decision and we don't foresee, at this moment, any reason to change it. However, this is not a very restrictive constraint, as a matter of fact all the major UNIX platforms as well as Microsoft platforms provide a functional JVM.

A more restrictive constraint that the current implementation of the framework imposes to computational resources is the requirement to support IP multicast communication. In fact, both the discovery-and-join protocol and the recovery-from-crash protocol utilize this type of communication. The first protocol implements the capability to search the Harness name space for an active DVM and enroll into it, while the second protocol is used to keep a consistent status in the event of host or network crashes. To relax this restriction we plan to develop a non-multicast version of these two protocols relying on centralized services for future releases of the software, however the current prototype requires the support for IP multicast.

A third constraint imposed by the framework is the requirement for plug-ins to appear to the system as Java classes. Anyway, this requirement implies very small restrictions on the way users may develop plug-ins, in fact a plug-in is not required to be a monolithic entity and may depend on any number of other classes. Besides, if efficiency requirements are too strict to be fulfilled by pure Java code, it is possible to exploit the JNI to either manually wrap native code into Java classes [7] or to exploit one of the tools that perform automatic wrapping of legacy code [8].

Harness provides a flat mapping of Internet host names ont computational resources names. However, the framework allows each user to adopt his own computational resource naming and grouping scheme as long as he is able to provide a plug-in to map that scheme onto the Internet host names space. Users willing to share a same name space need to use the same name-mapping service. This mechanism allows the definition of abstract user-defined computational resources naming policies without compromising the coherency of the underlying metacomputing framework. The same mechanism

The same mechanism has been adopted for plug-ins names. Thus, the framework provides a flat mapping of names of plug-in names onto Java class names, but each user can register his own grouping and naming policy providing his own mapper plug-in.

3.2 Harness implementation and protocols

The kernel level services of a Harness DVM are delivered by a distributed system composed of two categories of entities:

- a DVM status server, unique for each DVM;
- a set of Harness kernels, one and only one running on each computational resource currently enrolled or willing to be enrolled into a DVM.

To achieve the highest possible degree of portability for the kernel level services both the kernel and the DVM status server are implemented as pure Java programs. We have used the multithreading capability of the Java Virtual Machine to exploit the intrinsic parallelism of the different tasks the two entities have to perform, and we have built the framework as a set of Java packages.

Control messages and DVM status changes not related to the discovery-and-join protocol or the recover-from-failure protocol, are exchanged through a star shaped set of reliable unicast channels whose center is the DVM status server. These connections are implemented through the communication commodities delivered by the java.net package. It is important to notice that neither the star topology, nor the use of the java.net package are constraints imposed to all the communication services in the framework. On the contrary, user level communication services may adopt the connection topology that best suit their needs and are not required to use the java.net package to implement these commodities. For this reason, neither the star topology interconnecting the kernels and the DVM Server, nor the fact that the java.net package is used represent a major bottleneck in the Harness metacomputing framework. The kernels and the DVM server interacts to guarantee a consistent evolution of the status of the DVM both in front of users requesting new services to be added and in front of computational resources or network failures. This consistency is enforced

by means of a set of protocols executed during the different phases of the DVM life. In the following subsections we describe each of them

3.2.1 DVM startup protocol

A DVM may be started in three different ways:

- starting a DVM server;
- starting a kernel;
- starting an application.

In the first case, a user invokes the execution of the main method of the Java H_Server class from the edu.emory.mathcs.harness package providing as a parameter the name of the DVM this server is starting. The DVM server executes a hashing function to map the DVM name into a multicast IP address and port. Then it starts to multicast on the channel I'm alive packets and to listen for incoming packets. The DVM server can get three types of packets:

- I'm alive packets from a DVM server;
- Join packets from kernels;
- Query packets from applications.

The server checks the source address of any I'm alive packet it receives. If the packet comes from another server the server multicasts a train of I'm alive packets to notify its presence to the other server and then it exits. This will enforce the kernels running on computational resources enrolled in the DVM to start the server regeneration protocol and to regenerate a new, single server. This mechanism prevents the existence of multiple DVM servers with partial or outdated information and guarantees that a single DVM server is active in a DVM.

If the server receives a join packet then it generates a TCP connection to the sender kernel and it starts the Join protocol.

If the server receives a query packet then it checks if a kernel exists on the computational resource from which the application is querying. If a kernel is already active, then the server provides to the querying application the port number on which the kernel accepts connections from applications, otherwise it provides a null reply.

The second way to start a Harness DVM is to invoke the main method of the Main class in the edu.emory.mathcs.harness package providing as a startup parameter the name of the DVM the kernel wants to enroll into. The kernel executes the hash function to map the DVM name into an IP multicast address and port and sends send a Join packet on that channel. The kernel performs three tries before giving up. After three tries have timed out without a DVM server activating a TCP connections the kernel assume no DVM server exists and spawns a new JVM to start a new DVM server. Then he starts again sending the Join packet.

The third way to start a Harness DVM is to instantiate the class H_core or H_RMIcore from the package edu.emory.mathcs.harness in an application providing the DVM name as a parameter. The class constructor executes the hashing function and drops a query packet on the multicast channel. If no answer comes back or if the answer says that no kernel is active on the computational resource the constructor spawns a new JVM starting a kernel and sets a flag to avoid starting a new one even in the case of another failed set of tries. The possibility of two or more applications racing to spawn to or more kernels on the same computational resource is prevented by the Join protocol.

3.2.2 Join and leave protocols

The DVM server initiates the join protocol each time it receives a multicast join packet. The Join packet contains the IP address and a port number onto which the willing-to-join kernel is accepting a TCP connection. The first step of the join protocol is the instantiation of a TCP connection between the DVM server and the Joining kernel. Then the DVM server waits for the kernel to provide its baseline. At this point the server performs two checks: the baseline check and the uniqueness check. The baseline check consists of checking the compatibility of the kernel with the current implementation of the DVM server. The uniqueness check consists of checking that no other kernel has already joined from the same computational resource. In case of failure of one of these two checks an error message is sent back, the protocol terminates with a failure and the connection is closed. If the kernel passes both controls then the DVM servers checks if the kernel is Joining back after a failure (computational resource or network crash) or if the computational resource has never been enrolled in the DVM before. If the computational resource is coming back from a crash the DVM server sends to the kernel a crash token message and a copy of its pre-crash status, otherwise it sends a new token message. The following step is to get from the kernel its current status and to send back to it the current status of the DVM.

At this point the Join protocol is successfully completed, the DVM server generates a Join event that is distributed as described in next section while the kernel is now enrolled in the DVM.

The leave protocol is much simpler that the Join protocol. The leave protocol is always started by a kernel. A TCP connection between the kernel is guaranteed to be active, as a matter of fact it is not possible to start the Leave protocol before a successful completion of the Join protocol. The kernel sends an explicit Leave message to the DVM server and then closes the TCP connection. The DVM server generates a Leave event that is distributed as described in next section.

3.2.3 Totally ordered distribution of DVM status changes

The status of the DVM consists of the set of computational resources currently enrolled in the DVM, the set of services available on each enrolled computational resource as well as the DVM's baseline. We call baseline of a DVM the minimum set of services a computational resource must be able to deliver in order to join the DVM. The dynamic nature of the framework make this state an evolving entity, thus the framework keeps it up to date and available for queries from any application or service in the DVM. It is important to notice that information about the applications currently using services or internal status of an application is not part of the DVM status and loosing track of it does not in any way compromise the existence of the DVM in itself. Any form of application tracking and check-pointing, while highly desirable for many applications, is a service in itself and the framework does not need to incorporate it in its status.

The Harness metacomputing framework guarantees that all the events that changes the status of the DVM are received by all the kernels enrolled in the DVM in the same order. In the current implementation the Total Order (TO) protocol is implemented adopting the DVM server as a central ordering entity and exploiting the stream nature of TCP connections to avoid subsequent losses of order. Although very simple, a centralized implementation of the TO protocol has in general two negative features:

- the central entity is a single point of failure;
- the central entity is a bottleneck.

However, these two problems do not represent a major flaw in the design and efficiency of our framework. In fact, the single point of failure is limited to the incapability of the framework to retrieve after a DVM server crash the status of a previously crashed kernel and the central bottleneck does not influences application level communication services. The status of a DVM as it is defined in the Harness metacomputing framework consists of the sum of the stati of each enrolled kernel. Each event that changes the status of the DVM changes the status of a kernel in a way that is recorded by the kernel itself with the only exception being the case of a kernel crash. Thus it is not possible for an event, except for kernel crash events, to get lost in a DVM server crash. On the contrary, in the case of a DVM server crash it is possible to reconstruct completely the current status of the DVM simply obtaining from every surviving kernel a copy of its current status.

It is important to notice that the fact that this reconstruction process is not able to keep track of crashed kernels does not mean that applications relying on services delivered by the crashed kernels will have as their only choice to stop and fail. Reliable distributed check-pointing of application's status and restart of failing services are services themselves, thus their behavior in the event of kernel crashes is not constrained by the DVM status and the reconstruction of the DVM status is not concerned with them.

To evaluate the bottleneck represented by the star topology we have measured the performance of the following experimental setup. A SparcStation 5 running Solaris 5.6 hosted the DVM server, a Harness kernel and some other common unrelated applications (Internet browser, X server, etc.). Harness kernels were hosted by other, heterogeneous machines connected to the DVM server on a 10 megabit ethernet network. The average time required by DVM server to process and distribute events was 10 ms. We have repeated the measurements with an increasing number of enrolled kernels but the system showed only a negligible overhead (less then 10%) for up to 20 kernels. Although 10 ms is not a negligible amount of time, it is important to notice that it involves only events requiring DVM status changes, as a matter of fact any traffic generated by user application exchanging data is not required to flow through the DVM server. The only events that the DVM status server needs to process are:

- a kernel joining the DVM;
- a kernel leaving the DVM;
- a kernel crash;
- the addition of a service to the DVM.

Thus the DVM server represents only a marginal bottleneck in the Harness metacomputing framework.

3.2.4 The core library

Any computational resource enrolled in a Harness DVM is able to provide from the start only a single service: the capability to add services to the set of service currently available by loading plug-ins in a distributed, coordinated fashion. We call this capability the Basic Loading Service. Any additional service can be plugged-in on demand if a plug-in able to deliver it is available. This design allows the Harness system to be as open-ended as possible, as a matter of fact the only hard-wired service is the basic loading service and any other service may be developed at a later stage and added on demand by users requiring it.

The requirement that each plug-in appears to the system as a Java class allowed us to adopt the Java classes name space as the Harness Plug-in name space. This name space allows users to generate fully package-qualified names that are virtually collision-free. However, the Harness metacomputing framework guarantees that name collisions will not compromise the coherency of the programming environment. This guarantee is enabled by run-time checking the uniqueness of any plug-in loaded in the DVM.

The basic loading service is implemented through a loading protocol and delivered to users by means of a core

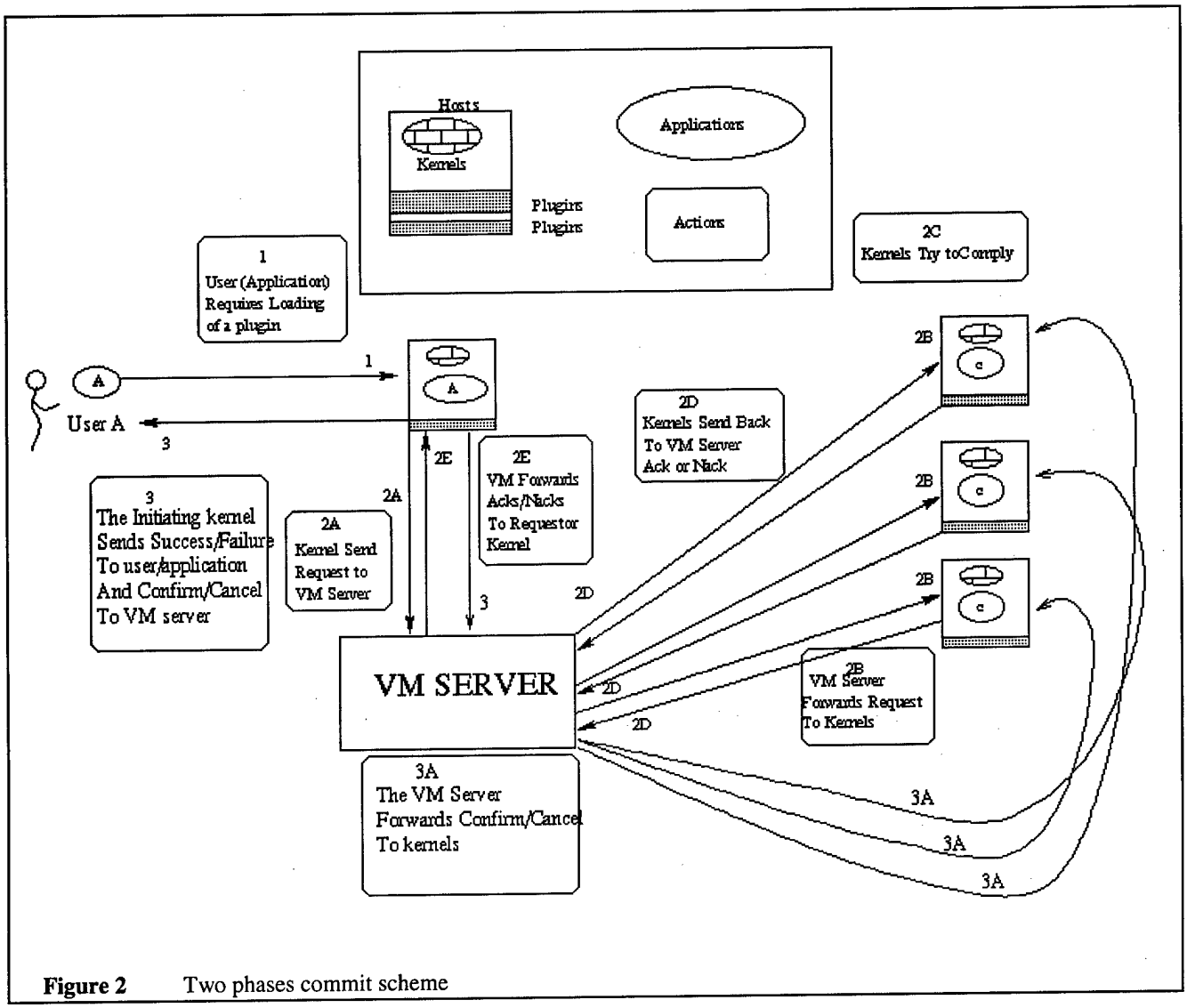


Figure 2 Two phases commit scheme

library. The core library is the main service access point for applications to a DVM. We have developed two different versions of the core library: a pure Java, object oriented, Remote Method Invocation (RMI) based one, and a generic, loosely typed, socket based one. The first version of the core library takes full advantage of the object oriented features of the Java programming language as well as of the Object Serialization mechanism and of the RMI capabilities. The second version of the core library has been designed to allow legacy code based applications and, more in general, non Java-based applications, to setup and adapt the environment by means of the basic loading service. In this second version of the library all the data exchanged between the application and the kernel have been demoted to be strings of characters and the core library takes care of marshalling and un-marshalling the parameters according to the requirements of the language of the application. Currently, only the object oriented pure Java version of the core library has been fully developed, however, a test implementation of the generic version has been developed in Java to test and debug the generic socket interface in the kernel.

The core library provides access to the only fixed service access point of the Harness system, namely the functions:

- `public H_RMIcore(String DVMName);`
- `public void HC_RegisterUser(String username, String password, H_pname serviceMapper, H_pname RCmapper);`
- `public H_RetVal HC_Load(H_pname theServiceName, H_crname[] theComputationalResourcesNames,`

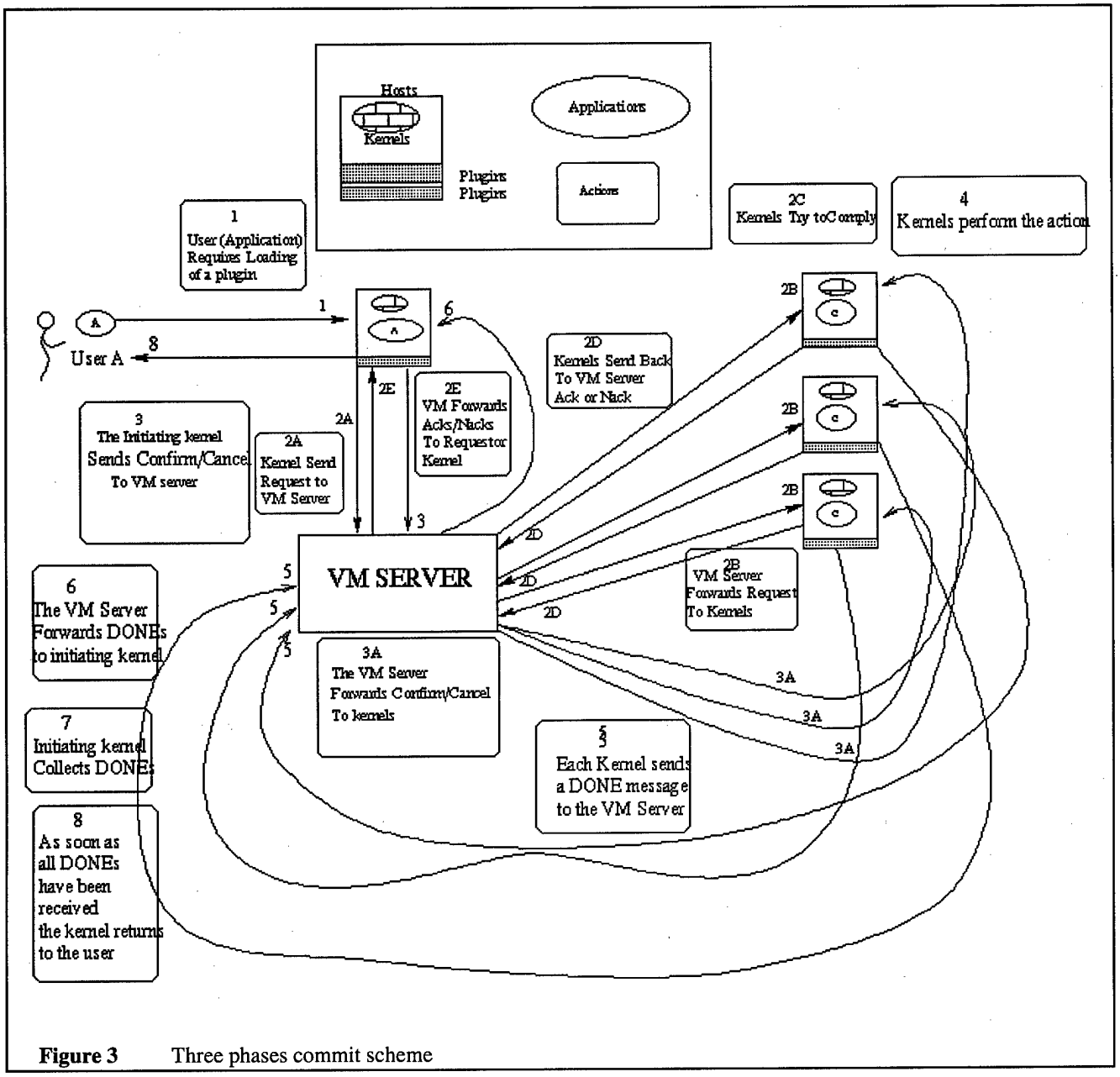


Figure 3 Three phases commit scheme

```
    H_QoS theQoS);
```

- `public H_RetVal HC_GetInterfaceDescriptor(H_Handle pluginHandle, H_CR computationalResource)`
- `public H_Info HC_GetInfo();`

The constructor requires a DVM name and performs the actions and control described in the section dedicated to the DVM startup sequence. The HC_RegisterUser function requires a username, a password, a service to plug-in mapper plug-in name and a computational resources to Internet hosts mapper plug-in name. If any of the mappers is null the system will user the default mapping for any other command otherwise the designated mapper is loaded and registered for the user. User names are associated to sets of capabilities. These sets of capabilities are local to each computational resource, thus each user can have different privileges granted on each computational resource enrolled in the DVM. By default a user name is associated to the set of capabilities of user nobody unless his user name is equal to the owner of the local Harness kernel. In the latter case he is associated

with set of capabilities of user root. The DVM stores the couples login name password and keep them for the whole life of the DVM. Thus a user can detach from the DVM and come back to a later time: as long as he is able to provide his password the DVM will grant him his set of capabilities.

The HC_Load function allows a user to add the named service on the named computational resources. If the user issuing the command has registered mappers for service and computational resources names those mappers will be used, otherwise the flat mapping will be performed. The operation is performed with the specified Quality of Service (QoS). The Harness computational framework supports four different QoSs that are generated as the combination of two parameters:

- all or none vs. at least one;
- two phases commit vs. completed execution guaranteed (three phases commit);

A load command issued with an all or none QoS will succeed if and only if all the required computational resources are able and willing to perform the operation. A load command issued with an at least one QoS will fail if and only if all the required computational resources are not able or not willing to perform the operation.

A load command issued with the two phases commit QoS will be performed according to the scheme described in figure 2. This QoS does not guarantee that at the time the call returns all the kernels have performed it, it merely guarantees that DVM status changing actions are not taken until they have been confirmed.

A load command issued with the completed execution guaranteed QoS will be performed according to the scheme shown in figure 3. This QoS guarantees that at the time a user command returns all the kernels have performed the requested action. A kernel failing to reply with the done message in reasonable time will cause the originating kernel to time-out and automatically generate a done message.

In both cases, time-outs guarantee that a user will not wait indefinitely for a command to complete.

3.2.5 The Harness classloader and the guaranteed uniqueness of loaded classes

In order to be able to retrieve classes from network repositories and from other computational resources enrolled in a DVM we developed a special classloader. This classloader performs several steps to look for a class enlarging at each step the scope of the search. These steps are:

- class loader cache;
- local file system;
- local file systems of any computational resource in the DVM;
- the whole set of repositories.

If, after these steps, the class loader has not found the required class then the loading process fails.

Although the loading and checking of new classes process is cumbersome and it requires distributed processing the framework executes it only when a class is referenced for the first time on a computational resource. Thus it does not represent a bottleneck for computation but merely a startup overhead.

The HC_GetInterfaceDescriptor function returns an instance of a class containing all the data necessary to an application to start using the service whose handle and providing computational resource have been passed as parameters.

The HC_GetInfo function returns an instance of the H_Info class. This class can be manipulated to get all the information about the status of the DVM. It is important to notice that an instance of the H_Info class is not an active entity, it is not automatically updated and it only reflects the status of the DVM at the moment it was created. On the other side the framework provides as a base service the capability to require automatic notification of DVM status changes as events and these events might be applied to the H_Info object to keep it up to date.

Each class that is loaded into a DVM, with the only excet\ption of the classes belonging to the standard Java distribution, is checked for uniqueness over the whole DVM. This check is performed using the DVM status server as the ordering authority of each class loading operation. Each time a class loader tries to load a class it issues the CRC32 of the byte array representing the class to the DVM status server. If the DVM server has stored a different value for the CRC32 of the named class the class loader will receive a deny message and the loading process will fail. If the class is not present yet in the DVM or if the CRC32 of the class that is being loaded is equal to the one registered by the DVM server the class loader is allowed to load the class and store its CRC in a local table. The DVM status server ensures that no races can occur, in fact it serializes all the checks. This centralized table does not represents a single point of failure, in fact, every computational resource stores locally every verified CRC. Thus, in case of a DVM status server crash, it is possible to reconstruct the table of all the CRC of all the classes loaded in the whole DVM by computing the union of the tables of all the computational resources enrolled in the DVM.

3.2.6 Reconstruction of the DVM status after a DVM status server crash: the regeneration protocol

The DVM status server issues periodic "I'm alive" messages on the DVM multicast channel. Each kernel checks the timely arrival of these messages to be sure the server is up and running. If a kernel misses two

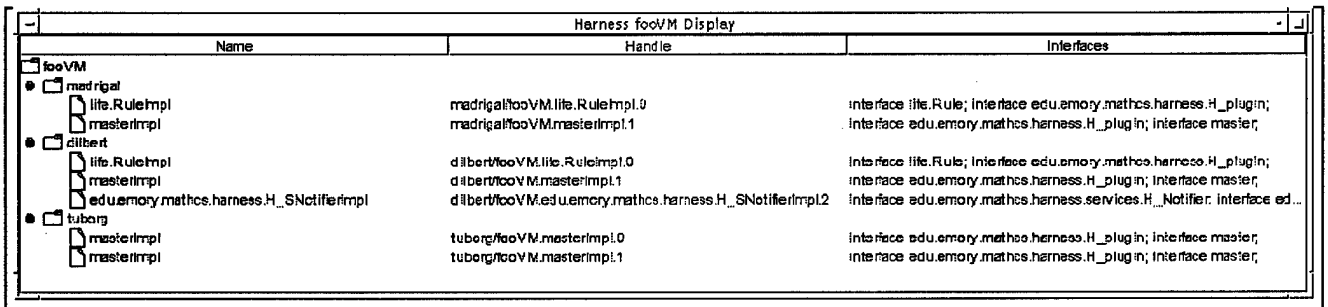


Figure 4 Snapshot of the Harness Distributed Virtual Machine Status Display application. The display shows every computational resource enrolled in the DVM, all the plug-ins currently loaded and the services each plug-in provides.

consecutive "I'm alive" messages it tries to ping the server on the reliable TCP connection, if there is no answer it starts the regeneration procedure. During the regeneration procedure applications can continue to use available services. However, no change to the DVM status is possible and the services cannot load new classes.

Each kernel starts the regeneration procedure by sending packets on the DVM multicast channel to notify to everyone else its presence. Each kernel compares the list of received packets received with the list of computational resources enrolled in the DVM at the time the crash occurred. After three rounds of packet exchanges every kernel selects the new candidate for the DVM status server according to a metric allowing no ties, the currently adopted metric is simply the lowest Internet address. The chosen candidate spawns a new DVM status server and every kernel joins back the DVM. When every surviving kernel has joined back the DVM the reconstruction of the status is complete with the only possible loss of the status of crashed kernels.

4 Example service, application and programming constraints

The single inheritance constraint imposed by the Java language makes the use of abstract classes extremely restrictive. As a matter of fact this constraint prevents users from taking advantage of capabilities provided by classes if those classes were not taken into account at the time the abstract class was defined. For this reason in the Harness metacomputing framework we have adopted Java Interfaces as the mechanism of choice to define the way in which applications can access services.

The execution of an application in the Harness metacomputing framework can be divided in two separate phases: environment set-up and actual execution. The setup phase can be further divided in three sub phases: **compatibility check**, **service loading** and **connection to service access points**. In the compatibility check sub-phase an application controls that the DVM is able to deliver the specialized services it needs. In the service-loading sub-phase the application requests the loading of the plug-ins that implement the services it needs. In this sub-phase the application can specify plug-ins names, thus posing constraints on the actual implementation of the services, or it can inquire the system for default plug-ins for the needed services. The default implementation of the service access point connection sub-phase consists of requesting to the DVM references to remote Java objects that will be accessed through Java RMI. However, Java RMI is not the only service access mechanism adopted in the Harness metacomputing framework. As a matter of fact legacy application can access legacy enabled Harness services through standards Internet sockets. This mechanism allows the porting of legacy applications to the Harness system by implementing the necessary plug-ins and the code for the setup phase.

Once the application has completed the environment setup it can start the actual execution phase. In this phase, the application accesses the services provided by the DVM through the service access points set up in the previous phase, thus it is completely independent from the service implementation.

As a first example of a Harness service we adopt the Synchronous DVM event notification service. This service consists of the capability for any application or other service to register as interested in DVM event and request to be notified of their occurrence. DVM events include the joining and leaving (or crashing) of any computational resource as well as the loading of any service. The synchronous DVM event notification service does not allow users to register callbacks, it requires users to actively request the next available event with a blocking call. A callback capable version of the service targeted to single-threads users can be easily layered on top of this service and implemented using the multi-threaded

```
1.  public class Display2
2.  {
3.      static public main(String[] argv)
4.      {
5.              h = new H_RMIcore(argv[0]);
6.              h.login(argv[1], argv[2]);
7.              HDVMDisplay theD = new HDVMDisplay();

8.              try
9.              {
10.                     H_crname[] targets = new H_crname[1];
11.                     targets[0] = new H_crname(InetAddress.getLocalHost().getHostName());
12.                     retval    =    h.load(new    H_pname("edu.emory.mathcs.harness.H_SNotifierImpl"),
    targets, new H_QoS("ALL"));
13.                     H_Notifier theN = (H_Notifier)retval.references[0];
14.                     myID = myN.H_register();
15.                     H_Info theInfo = h.getInfo();
16.                     theD.setCurrent(theInfo);

17.                     while(true)
18.                     {
19.                             theD.update(theN.getNext());
20.                     }
21.             }
22.             catch(Exception e)
23.             {
24.                     System.err.println(e);
25.             }
26.     }
27. }
```

Figure 5 Java code for the main class of the Harness Distributed Virtual Machine Status Display application.

architecture of the JVM.

The Harness metacomputing framework defines two Java interfaces related to the synchronous event notification service: the H_Notifier interface and the H_INotifier interface. The first one defines the set of functions users can use to interact with the service. The second Java interface is the one that a plug-in needs to implement in order to have the system dispatch the events to it. The plug-in guarantees that all the DVM events occurring in the DVM after the user registration will be queued exactly in the order in which they occurred and delivered in that order on demand.

We have used this service to develop a simple DVM status display application. In figure 4 you can see a snapshot of the display offered by this application, while figure 5 shows part of the application code. Lines 5 to 13 represents the set-up phase, while the actual execution phase consists of lines 14 to 20.

As any Harness application the display need first to instance a new Harness core library in order to connect to a DVM (see line number 5). Then it logs into the DVM providing a user name and a password (see line 6). In line 7 the applications generates an instance of a graphical status display class. This application needs no specialized services, thus there is no compatibility check sub-phase. In line 10 to 12 the application executes the services loading sub-phase, by loading a plug-in implementing the notifier service into the DVM. The service access points connection sub-phase consists of line 13 alone, in fact the application retrieves a reference to the service and stores it into a variable.

In line 14 the application registers itself as a recipient for system events, then in line 15 gets the current status of the system and in line 16 sets it into the display. Lines 17-20 loop endlessly to get new events and update the display accordingly.

This simple application clearly shows the main characteristics of Harness applications:

- the independence of the application code from the service implementation;
- the clear separation of set-up phase from execution phase.

5 Related works

Metacomputing frameworks have been popular for nearly a decade, when the advent of high end workstations and ubiquitous networking in the late 80's enabled high performance concurrent computing in networked environments. PVM [9] was one of the earliest systems to

formulate the metacomputing concept in concrete virtual machine and programming-environment terms, and explore heterogeneous network computing. PVM is based on the notion of a dynamic, user-specified host pool, over which software emulates a generalized concurrent computing resource. Dynamic process management coupled with strongly typed heterogeneous message passing in PVM provides an effective environment for distributed memory parallel programs. PVM however, is inflexible in many respects that can be constraining to the next generation of metacomputing and collaborative applications. For example, multiple DVM merging and splitting is not supported. Two different users cannot interact, cooperate, and share resources and programs within a live PVM machine. PVM uses Internet protocols which may preclude the use of specialized network hardware. The Harness "plug-in" paradigm effectively alleviates these drawbacks while providing greatly expanded scope and substantial protection against both rigidity and obsolescence.

Legion [10] is a metacomputing system that began as an extension of the Mentat project. Legion can accommodate a heterogeneous mix of geographically distributed high-performance machines and workstations. Legion is an object oriented system where the focus is on providing transparent access to an enterprise-wide distributed computing framework. As such, it does not attempt to cater to changing needs and it is relatively static in the types of computing models it supports as well as in implementation.

Globus [11] is a metacomputing infrastructure which is built upon the "Nexus" [12] multi-language communication framework. The Globus system is designed around the concept of a toolkit that consists of the pre-defined modules pertaining to communication, resource allocation, data, etc. However the assembly of these modules is not supposed to happen dynamically at run-time as in Harness. Besides, the modularity of Globus remains at the metacomputing system level in the sense that modules affect the global composition of the metacomputing substrate.

Sun Microsystems Jini project [13] presents a model where a federation of Java enabled objects connected through a network can freely interact and deliver services to each other and to end users. In principle Jini shares with Harness many keywords and goals, such as services as building blocks and hetereogeneity of service providers. However, Jini focuses on the capability to build up a world of plug-and-play consumer devices and, to cope with such a goal, increase the resolution of the computational resources that can be enrolled in a Jini federation. In fact in the Jini model these resources range from complete computational systems down to devices such as disks, printers, TVs and VCRs.

The CORBA Object Management Architecture [14] provides a model to which Object Requests Brokers of different vendors can refer in order to seamlessly interact, this generality is achieved defining protocols and interfaces to be used by Object Request Brokers. However the focus of CORBA is upon interoperability of service providers while the focus of our project is mainly on building, managing and dynamically reconfiguring such service providers. Besides, a large part of the CORBA system is dedicated to overcome the problems related to have a multi-language system, while Harness completely delegates this issues to Java and the JNI.

The CONDOR [15] project has demonstrated the usefulness of a flexible approach to the problem of resource gathering. However, the CONDOR project does not envision dynamic reconfigurability of the set of services provided by the computational resources.

Almost all the above projects envision a model in which very high performance bricks are statically connected to build a larger system. One of the main idea of the Harness project is to trade some efficiency to gain enhanced global availability, upgradability and resilience to failures by dynamically connecting, disconnecting and reconfiguring heterogeneous components. Harness is also seen as a research tool for exploring pluggability and dynamic adaptability within DVMs.

6 Concluding remarks

In this paper we have described the Harness metacomputing framework. The main feature of this system is its capability to allow users to build, reconfigure, use and dismantle Distributed Virtual Machines (DVM). Harness dynamically enrolls and reconfigures heterogeneous computational resources into DVMs in order to provide a consistent service baseline to the users. This fundamental characteristic of Harness is intended to address the inflexibility of current metacomputing frameworks as well as their incapability to incorporate new technologies and avoid rapid obsolescence. These results are achieved without compromising the coherency of the programming environment by means of a distributed plug-in mechanism, a high level of code portability and a fault-tolerant dynamic status management mechanism.

In this paper we have shown that the current prototype of Harness is able:

- to define services in term of abstract Java interfaces in order to have them updated in a user transparent manner;
- to adapt to changing user needs by adding new services to heterogeneous computational resources via the plug-in mechanism;
- to cope with network and hosts failures with a limited amount of overhead;

Although the model for security and access control has

been defined and briefly described in this paper, the current prototype does not implement it. This feature will be included in future releases.

7 References

1 N. Boden et al., MYRINET: a Gigabit per Second Local Area Network, IEEE-Micro, Vol, 15, No. 1, February 1995.

2 S. Pakin, M. Lauria and A. Chien, High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet, Proc. of the 1995 ACM/IEEE Supercomputing Conference, December 3-8, 1995, San Diego, CA, USA.

3 T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.

4 Sun Mycrosystems, Remote Method Invocation Specification, available on line at http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html, July 1998.

5 Sun Mycrosystems, Object Serialization Specification, available on line at http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html

6 S. Liang, The Java Native Interface: Programming Guide and Reference, Addison Wesley, 1998.

7 V. Getov, S. Flynn-Hummel and S. Mintchev, High Performance Parallel Programming in Java: Exploiting Native Libraries, to appear on Concurrency: Practice and Experience, 1998.

8 S. Mintchev and V. Getov, Automatic Binding of Native Scientific Libraries to Java, Proceeding of ISCOPE97, December, 1997.

9 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam, PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.

10 A. Grimshaw, W. Wulf, J. French, A. Weaver and P. Reynolds. Legion: the next logical step toward a nationwide virtual computer, Technical Report CS-94-21, University of Virginia, 1994.

11 I. Foster and C. Kesselman, Globus: a Metacomputing Infrastructure Toolkit, International Journal of Supercomputing Application, May 1997.

12 I. Foster, C. Kesselman and S. Tuecke, The Nexus Approach to Integrating Multithreading and Communication, Journal of Parallel and Distributed Computing, 37:70-82, 1996

13 Sun Microsystems, Jini Architecture Overview, available on line at http://java.sun.com/products/jini/whitepapers/architectureoverview.pdf, 1998.

14 J. Siegel, CORBA Fundamentals and Programming, John Wiley & Sons Inc., New York, 1996

15 M. J. Litzkow, M. Livny and M. W. Mutka, *Condor - A Hunter of Idle Workstations*, Proc. of the 8th International Conference on Distributed Computer Systems, pp. 104-111, IEEE Press, June 1988.

Biographies

Mauro Migliardi is born in Genova (Italy) in 1966.
He got a Laurea degree in Electronic Engineering from the University of Genova in 1991 and a PhD in Computer Engineering from the University of Genova in 1995.
He is currently a research associate at Emory University.
His main research interests are parallel, distributed, heterogeneous computing and
metacomputing and high performance networking.

Vaidy Sunderam is Professor of Computer Science at Emory University, Atlanta, USA.
His current and recent research focuses on aspects of distributed and concurrent computing in heterogeneous networked environments. He is one of the principal architects of the PVM system, in addition to several other software tools and systems for parallel and distributed computing.
He has received several awards for teaching and research, including the IEEE Gordon Bell prize for parallel processing.
Recently his research activities have included novel techniques for multithreaded concurrent computing, input-output models and methodologies for distributed systems, and integrated computing frameworks for collaboration.

Parallel C++ Programming System on Cluster of Heterogeneous Computers

Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Shinji Sumimoto,
Toshiyuki Takahashi, and Hiroshi Harada
Real World Computing Partnership
{ishikawa, hori, tezuka, s-sumi, tosiyuki, h-harada}@rwcp.or.jp

Abstract

A parallel programming system, called MPC++, provides parallel primitives such as a remote function invocation, a global pointer, and a synchronization structure using the C++ template feature. The system has run on a cluster of homogeneous computers. In this paper, the runtime system is extended to run on a cluster made up of heterogeneous computers environment. Unlike other distributed or parallel programming systems on heterogeneous computers, the same program on the homogeneous environment runs on the heterogeneous environment in this extension.

1. Introduction

Most parallel programming language systems support not only communication between computers but also support remote function invocation and other parallel primitives. If such a system is designed to run on homogeneous as well as a heterogeneous computers environment, some issues, e.g., data conversion and the address of a remote function, must be solved. There are two approaches: One is to provide a parallel programming library. Whenever a remote function invocation is used, libraries of data conversion and marshalling arguments are called, followed by calling a remote function invocation library. Since such libraries are called at every remote function invocation, regardless of whether or not the receiver is the same machine type, it has an overhead when the sender and receiver are the same machine type.

The other one is to design a new parallel programming language. This supports good programming facilities, but requires the building of a compiler to generate efficient code for both the homogeneous and heterogeneous environments. However, the users need to learn the new language, which is usually not accepted.

MPC++ provides parallel primitives such as remote function invocation, a global pointer, and a synchronization structure using the C++ template feature without any extensions to C++. The system assumes an SPMD programming model where the same program runs but uses different data on computers. The runtime system is extended to run on the heterogeneous computers environment without changing any existing MPC++ library specifications. The MPC++ new runtime system solves disadvantages of the library approach to supporting both the homogeneous and heterogeneous environments.

In the following sections, an overview of MPC++ is first described in section 2. Section 3 briefly discusses issues on the heterogeneous environment. The design and implementation in the heterogeneous environment is presented in section 4. Section 5 presents preliminary evaluation results using two Sun Sparc Station 20s, one Intel Pentium, and one Compaq Alpha. Related work is described in section 6. Finally, we conclude this paper in section 7.

2. MPC++

The MPC++ program is assumed to run on a distributed memory-based parallel system. Program code is distributed to all physical processors and a process for the program runs on each processor.

To support parallel description primitives in MPC++ Version 2.0, the MPC++ multiple threads template library (called MTTL in short), realized by C++ templates, has been designed and implemented[5]. It contains i) `invoke` and `ainvoke` function templates for synchronous and asynchronous local/remote thread invocation, ii) `Sync` class template for synchronization and communication among threads, iii) `GlobalPtr` class template for pointer to remote memory, and iv) `yield` function to suspend thread execution and yield another thread execution. In this paper, the invocation and global pointer mechanisms are described.

2.1. invoke/ainvoke

The `invoke` function template allows us to invoke a remote function which involves creation of a new thread on

0-7695-0107-9/99 $10.00

the remote processor.

The invoke function template has two formats, one for a function returning a value and one for a void function. The former invoke format takes i) a variable where the return value is stored, ii) the processor number on which a function is invoked, iii) a function name, and iv) its arguments. The latter invoke takes i) the processor number on which a function is invoked, ii) a function name, and iii) its arguments.

The following example shows that a foo function is invoked on processor 1. The execution of the mpc_main thread is blocked until foo function execution is terminated. After the end of foo function execution, the return value is stored in variable i and then mpc_main thread execution is resumed. A void function is invoked on processor 2 in line 9. After the execution of the bar function is finished, mpc_main thread execution is resumed.

```
 1 #include <mpcxx.h>
 2 int      foo(int, int);
 3 void     bar(int, int);
 4 mpc_main()
 5 {
 6     int          i;
 7
 8     invoke(i, 1, foo, 1, 2);
 9     invoke(2, bar, 10, 20);
10 }
```

The ainvoke function template is provided to program asynchronous remote/local function invocation. Asynchronous remote function invocation means that a thread invokes a remote function and executes the subsequent program without blocking. The thread may get the return value later using the synchronization structure.

2.2. GlobalPtr

Any local object can be referred to using a global pointer which is realized by the GlobalPtr class template. The GlobalPtr class template takes one type parameter which represents the type of the storage pointed to by the global pointer. The operations on an GlobalPtr object are almost the same as a regular pointer object except that a global pointer of a global pointer is not allowed.

A simple example is shown below. The foo function takes a global pointer and a value as the parameters and saves the value into the storage pointed to by the global pointer. The remote foo function is invoked in line 16. A global pointer gp is initialized in 15 where the address of g1 on processor #0 is set. In line 17, a global pointer gp points to the address of g1 on processor #2 using the set method of the GlobalPtr object.

```
 1 #include <mpcxx.h>
 2 int                          g1;
 3 void foo(GlobalPtr<int> gp, int val)
 4 {
 5   *gp = val;
 6 }
 7 void bar(GlobalPtr<int> gp)
 8 {
 9   printf("[Processor %d] *gp = %d\n",
10          myNode, (int) *gp);
11 }
12 mpc_main(int, char**)
13 {
14   GlobalPtr<int>          gp;
15   gp = &g1;
16   invoke(1, foo, gp, 10);
17   gp.set(&g1, 2);
18   *gp = 20;
19   printf("[Processor %d] g1 is %d\n",
20          myNode, g1);
21   invoke(2, bar, gp);
22 }
```

When the example is executed, you see the following message:

```
[Processor 0] g1 is 10
[Processor 1] *gp = 20
```

Note that the GlobalPtr has the nwrite and nread functions to write/read data to/from a remote node.

3. Issues

Issues in the heterogeneous computers environment are briefly reviewed below.

1. Data Type and Representation
   Of course, data type and its representation are different on heterogeneous computers, e.g., little endian vs. big endian. A *long* value represents 64 bits or 32 bits depending on processor type.

2. Function Address and Global Scope Data Address
   The MPC++ assumes the SPMD programming style where the same program runs on each computer. It means that the addresses of a function and global scope data are the same locations over computers in the homogeneous environment. Using this feature, the MPC++ realizes a light weight function invocation and global pointer mechanisms. On the other hand, though the program is compiled on all machines on the heterogeneous environment, the addresses of a function and global scope data are different locations over computers.

3. Global Pointer
   A global pointer enables access to a remote memory location across different processor types. If a pointer points to a data structure, the representation of that structure is different between computers.

4. Design and Implementation

4.1. Remote Function

A remote function invocation mechanism in a distributed memory-based parallel computer is usually realized as follows:

1. The Sender:
   An invocation function performs the following procedure:
   - A function address and its arguments are packed into a request message, known as *marshalling*.
   - The request message is sent to the receiver.
   - The reply message is received and the result is stored.
2. The Receiver:
   The receiver has a dispatcher which performs the following procedures:
   - A request message is unpacked, known as *unmarshalling*. To unmarshal the message, the dispatcher needs to know the data types of arguments and a function's return value.
   - A function, whose address is defined in the message, is invoked.
   - The return value is sent back to the sender.

The dispatcher on the receiver can not be straightforwardly implemented. Of course, in the homogeneous computers environment, the dispatcher receives the size of arguments and a return value so that it creates a call frame without knowing data types. This implementation usually requires assembler code, and thus the code is not portable.

Our approach is that a dispatcher for each request message type is created so that the dispatcher knows all data types. In the rest of this chapter, first, class and function templates are introduced to realize a remote function invocation mechanism in the homogeneous computers environment. Then these templates are modified to adapt to the heterogeneous computers environment.

4.1.1. Homogeneous Computers Environment

First of all, a request message data structure is defined by the C++ template feature. The following example defines a request message for two arguments where a function address and its argument are contained:

```
1 template<class F, class A1, class A2>
2 struct req_message2 {
3     F   (*func)(A1, A2);
4     A1  a1;
5     A2  a2;
6 };
```

```
 1 template<class F, class A1, class A2>
 2 class _dispatcher2 {
 3 public:
 4  static void invoker() {
 5    req_message2<F, A1, A2> *argp;
 6    F                       val;
 7    argp = (req_message2<F, A1, A2> *)
 8                  _getArgPrimitive();
 9    val = argp->func(argp->a1, argp->a2);
10    _remoteReturnPrimitive(&val,
11                           sizeof(val));
12    }
13 };
```

Figure 1. A dispatcher in the homogeneous environment

```
 1 template<class F, class A1, class A2>
 2 inline void
 3 invoke(F &ret, int pe, (F (*f)(A1, A2)),
 4        A1 a1, A2 a2) {
 5  req_message2<F, A1, A2>        arg;
 6  arg.func = f;
 7  arg.arg1 = a1;
 8  arg.arg2 = a2;
 9  remoteSyncInvokePrimitive(pe,
10         _dispatcher2<F, A1, A2>::invoker,
11         &arg, sizeof(arg), &ret);
12 }
```

Figure 2. `invoke` template function in the homogeneous environment

The dispatcher for the above request message on the receiver is defined using the C++ template as shown in Figure1. In this example, a message is obtained by the `_getArgPrimitive` runtime routine, and then a function is invoked using the function address specified by the message. The return value is sent to the sender by issuing the `_remoteReturnPrimitive`.

An `invoke` function for a two argument function is defined in Figure 2. A message is constructed using the `req_message2` template, and then the `remoteSyncInvokePrimitive` is invoked to send the message to the sender specified by `pe`. When the receiver receives the message, the instantiated function of `invoker` of class `_dispatcher2<F, A1, A2>` is invoked.

It should be noted that templates `req_message2` and `_dispatcher2` are instantiated when the `invoke` template is instantiated. An example of the `invoke` template function usage is shown below and its compiled code is shown in Figure 3.

```
 1 extern int foo(int, double);
 2 mpc_main()
 3 {
 4     int         ret;
 5     req_message2<int, double>   arg;
 6     arg.func = f;
 7     arg.arg1 = a1;
 8     arg.arg2 = a2;
 9     remoteSyncInvokePrimitive(pe, _dispatcher2<int, double>::invoker,
10                               &arg, sizeof(arg), &ret);
11 }
12 struct req_message2 {
13     int         (*func)(int, double);
14     int         a1;
15     double      a2;
16 };
17 class _dispatcher2 {
18 public:
19     static void invoker() {
20         req_message2<int, double>
21                     *argp = (req_message2<int, double> *) _getArgPrimitive();
22         int                   val;
23         val = argp->func(argp->a1, argp->a2);
24         _remoteReturnPrimitive(&val, sizeof(val));
25     }
26 }
```

Figure 3. An instantiation of req_message2 and _dispatcher2

```
1 extern int foo(double);
2 mpc_main()
3 {
4     int         ret;
5     invoke(ret, 1, foo, 5, 2.0);
6 }
```

An invoke template function call in Line 5 of the above example is expanded to lines 5 to 10 in Figure 3. This expansion involves the instantiation of the `req_message2<int, double>` class and `_dispatcher2` class shown in the figure.

4.1.2. Heterogeneous Computers Environment

To adapt the runtime system to the heterogeneous environment, an initialization routine is introduced, and then the templates described before are extended. Figure 4 is used as a heterogeneous environment example which consists of two Sun Sparc machines running SunOS which are designated as PEs 0 and 1, one Intel Pentium machine running Linux referred to as PE 2, and one Compaq Alpha machine running Linux referred to as PE 3.

. Initialization and Primitives

An executable file has not only code but also has a symbol name table in which function and global data symbols and their addresses are stored. When a process of a MPC++ program is spawned on each computer, each process reads the executable file in order to read all symbols information and set up a name table for the function and global scope symbols and their addresses. Then, the processor type is passed to the other remote processes so that each process has the processor type table as shown in Figure 4.

Let us define three functions `_isHomoEnv`, `_marshalName`, and `_unmarshalName`. The `_isHomoEnv` function, whose argument takes a processor number, returns boolean *true* if the computer specified by the argument is of the same processor environment as the local processor. The `_marshalName` function takes two arguments, a buffer address and a function address or a global scoped data address. It stores the symbol name of the address into the buffer. The `_unmarshalName` function takes a buffer address which has a symbol name. It returns the address of the symbol name registered in the name table.

. invoke template function

The `invoke` template function shown in Figure 2 is modified for the heterogeneous environment as shown in Figure 5. If the receiver is the same processor type, the homogeneous invocation mechanism is used in line 4. Otherwise, the heterogeneous invocation mechanism is performed as shown in lines 6 to 15.

Unlike the homogeneous case, a dispatcher function for the heterogeneous environment is also converted to the symbol name in line 9. The dispatcher `hinvoker` function of

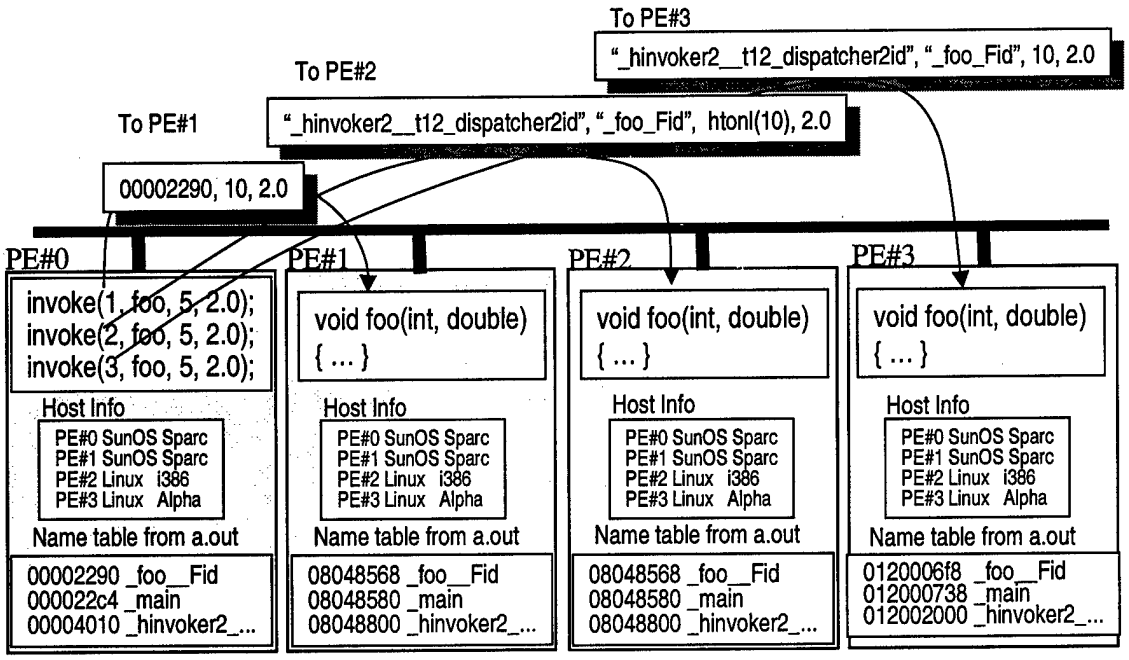


Figure 4. An Execution Environment Example

```
 1 template<class F, class A1>
 2 inline void invoke(F &ret, int pe, (F (*f)(A1)), A1 a1, A2 a2) {
 3     if (_isHomoEnv(pe)) {
 4         // Same as in Figure 2
 5     } else {
 6         char    buf[MAX_REQMSG];
 7         char    *argp;
 8
 9         argp = _marshalName(buf, _dispatcher2<F, A1, A2>::hinvoker);
10         argp = _marshalName(argp, f);
11         argp = _marshal(argp, a1);
12         argp = _marshal(argp, a2);
13         remoteSyncInvokePrimitive(pe, 0,
14                                   buf, argp - buf, &buf);
15         _unmarshal(buf, val);
16     }
17 }
```

Figure 5. `invoke` **template function in the heterogeneous environment**

a class template _dispatcher1 will be shown later. After marshalling a symbol name of the user function, arguments a1 and a2 are marshaled using the _marshal function.

The _marshal function is an overloaded function. The MPC++ runtime system defines all C++ basic data types. An example of prototype specifications is shown below. If an argument is a user defined data type or structure, its marshal operation must be defined.

```
1 caddr_t _marshal(caddr_t, int);
2 caddr_t _marshal(caddr_t, double);
```

As shown in line 13 of Figure 5, the second argument of remoteSyncInvokePrimitive is zero so that the function knows this call is for the heterogeneous environment.

The definition of hinvoker is shown in Figure 6. The hinvoker i) unmarshals all arguments, ii) invokes a function, iii) marshals the return value, and iv) sends a reply message to the sender. The _unmarshal function is an overloaded function, the same as the _marshal function.

As an example shown in Figure 4, a remote function invocation from PE#0 to PE#1 is the same mechanism as that the homogeneous environment since those two processors are the same processor and operating system. In the case of a remote function invocation from PE#0 to PE#2, since the sender knows that the remote processor's integer data type representation is different, the integer data is converted to the network byte order in addition to converting the addresses of the dispatcher function and user function foo to symbols. On the other hand, in the case of a remote function invocation from PE#0 to PE#3, the integer data is not converted since the integer data byte order of both is the same.

4.2. Global Pointer

In this subsection, a Global Pointer in the heterogeneous environment is implemented first, and then it is extended to adapt to the heterogeneous computers environment.

4.2.1. Homogeneous Computers Environment

Figure 7 shows that an implementation of a GlobalPtr class which is a simplified version of the actual MPC++ implementation. This implementation is described with the usage of a GlobalPtr object shown as follows:

```
1 int             gi;
2 void foo()
3 {
4     GlobalPtr<int> gp;
5     gp = &gi;
6     i = *gp;
7     *gp = 10;
8 }
```

Table 1. Evaluation Environment

Nodes	Two Sun Sparc Station 20s (75 MHz, 64MB, Sun OS 4.1.3) One Intel Pentium Pro (200 MHz, 128MB, NetBSD 1.2.1) One Compaq PWS 600au (Alpha 21164, 500 MHz, 128 MB, Linux 2.0.32)
Network	Myricom Myrinet
Runtime System	SCore standalone system with PM communication library

In line 5, an overloaded assignment function defined in line 19 of Figure 7 is invoked so that the address of gi and the processor number are stored in the internal data structure Gstruct of the GlobalPtr object. The assignment expression in line 6 has the same effect as the following expression:

```
i = (int)*gp;
```

This means that the cast operation to the integer type is invoked, led by the pointer reference operation. As defined in line 20 of Figure 7, the pointer reference operation returns the Gstruct object. The cast operation to the integer of Gstruct object is defined in line 8 of Figure 7.

4.2.2. Heterogeneous Computers Environment

In order to adapt the GlobalPtr facility to the heterogeneous computers environment, three cases are considered:

1. The _marshal and _unmarshal functions for the GlobalPtr class are defined so that a GlobalPtr object may be passed to a remote computer.
2. An assignment of a pointer address for remote data in the set method is extended so that the remote address is obtained. This requires extra communication.
3. The remote read/write operations are extended so that data conversion is performed for a different processor architecture.

5. Preliminary Evaluation Results

5.1. Environment

The MPC++ new runtime system has run on a heterogeneous computers testbed where two Sun SS20s, one Compaq PWS600au, and one PC are connected by the Myrinet network as shown in Table 1. The gnu egcs compiler is used to compile benchmark programs.

```
 1 template<class F, class A1, class A2>
 2 class _dispatcher2 {
 3 public:
 4     static void invoker() { ... }
 5     static void hinvoker() {
 6         caddr_t         argp = _getArgPrimitive();
 7         F               (*f)(A1, A2);
 8         A1              a1;
 9         A1              a2;
10         F               val;
11         char            buf[sizeof(F)];
12
13         argp = _unmarshalName(argp);
14         argp = _unmarshal(argp, a1);
15         argp = _unmarshal(argp, a2);
16         val = f(a1, a2);
17         argp = _marshal(buf, val);
18         _remoteReturnPrimitive(buf, sizeof(val));
19     }
20 };
```

Figure 6. A dispatcher in the heterogeneous environment

```
 1 template<class T>
 2 class GlobalPtr {
 3  private:
 4    class Gstruct {
 5    public:
 6      int       pe;
 7      caddr_t   addr;
 8      operator  T() {
 9           T buf;
10           _remoteMemRead(pe, addr, &buf, sizeof(T));
11           return buf; }
12      }
13     void operator=(T t) {
14           _remoteMemWrite(pe, addr, &t, sizeof(T));
15     }
16    };
17    Gstruct      gval;
18  public:
19    GlobalPtr<T> &operator=(T *data) {gval.pe = myNode; gval.addr = (caddr_t)data; }
20    GlobalPtr<T>::Gstruct &operator*() { return gval; }
21    void nwrite(T *data, int nitem);
22    void set(void *addr, int pn) { ... }
23 };
```

Figure 7. An Implementation of GlobalPtr

Table 2. Remote `void` zero arguments function cost (round trip time)

Sender to Receiver	RTT (μseconds)
Sun to Sun	36.0
Sun to Pentium	38.8
Sun to Alpha	36.6
Pentium to Sun	46.5
Pentium to Alpha	30.5
Alpha to Sun	44.7
Alpha to Pentium	30.5

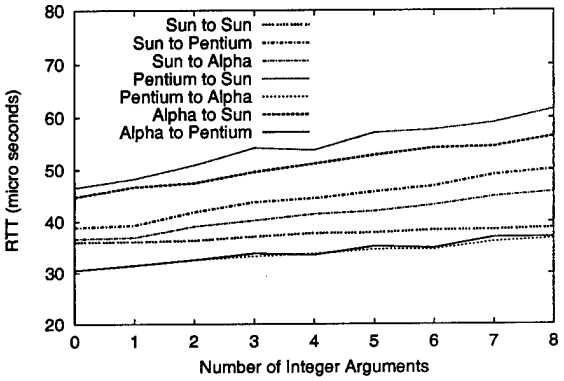


Figure 8. Remote Function Invocation

5.2. Remote Function Invocation Cost

The remote `void` no arguments function invocation cost, round trip time, is shown in Table 2. Figure 8 shows the cost of remote void function invocation whose arguments vary from 0 to 8.

According to these results, the cost of both Pentium to Sun and Alpha to Sun is greater than the one of Sun to Pentium and Sun to Alpha, i.e. about 8 micro second extra cost is added. Thus, we have further investigated the detailed cost analysis.

The cost of marshalling/unmarshalling of integers is less than two micro seconds as shown in Figure 9. Figure 10 shows the `_marshalName` and `_unmarshalName` costs when the `_dispatcher2<F, A1, A2>::hinvoker` function symbol is passed. Those function costs depend on the symbol name length because of making the hash key for the function name table and comparison of strings. As shown in this figure, the `_unmarshalName` cost on Sun is greater than others. This leads to an extra cost.

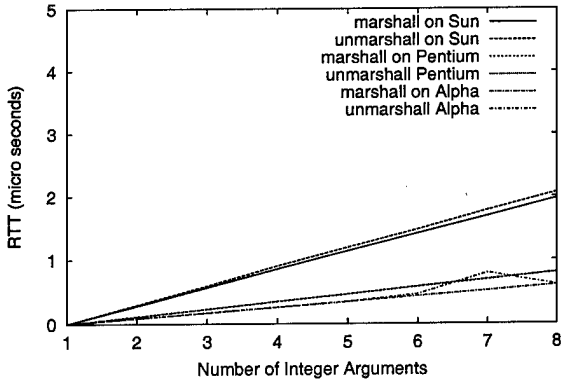


Figure 9. Integer Data Marshalling/Unmarshalling Cost

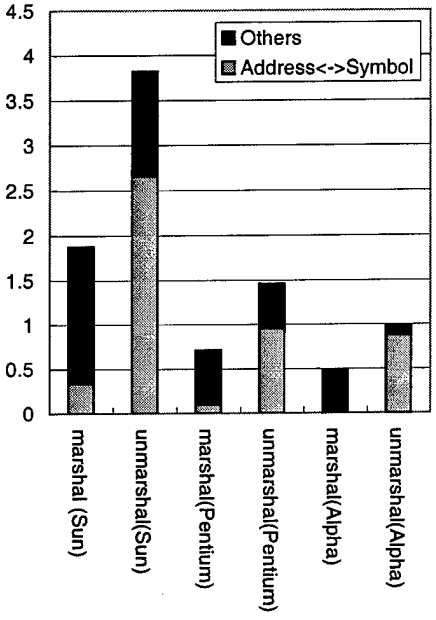


Figure 10. Function Address Marshalling/Unmarshalling Cost

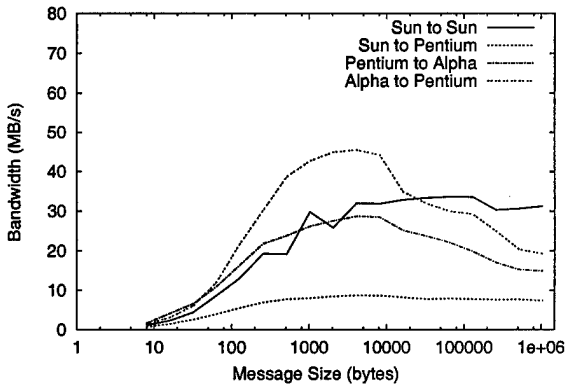


Figure 11. Remote `double` Data Write

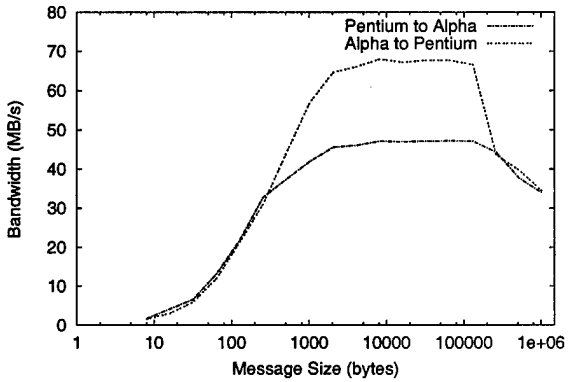


Figure 12. Revised Remote `double` Data Write

5.3. Remote Memory Write Cost

Figure 11 shows the bandwidth of a remote memory write whose area contains a `double` data type using the `nwrite` method of the `GlobalPtr<double>` object. The bandwidth between different processors is low due to the data conversion cost.

As shown below, the C++ template allows us to write a special `nwrite` method for the `double` data type so that no conversion is required between Pentium and Alpha machines. Figure 12 shows the result of this improvement.

```
1 GlobalPtr<double>::nwrite(double *data,
2                           int nitem)
3 {
4   /* special method for double */
5 }
```

6. Related Work

The template technique for the function invocation and global pointer described in this paper has also been realized by ABC++[8] and HPC++ Lib[2]. As far as we know, the ABC++ runtime only supports the homogeneous computers environment. The HPC++[2] provides a registration mechanism for a remote function invocation in the heterogeneous environment. This means that a program running in the homogeneous environment differs from a program running in the heterogeneous environment.

Though the HPC++ does not assume the SPMD execution model in the heterogeneous environment, the technique described in this paper can be applied as follows: Instead of the registration, a function defined in another program is defined with a dummy body so that the function address and its symbol are locally defined. Of course, the function address is meaningless except that the address is used to search the symbol name locally. The symbol name is passed to the receiver where the symbol is mapped to a function address in the receiver.

CC++[7] is a language extension to C++ such that global pointers, synchronization structures, and other parallel constructs are introduced. The CC++ runtime system has run in the heterogeneous computers environment using the Globus runtime system[1]. MPC++ realizes the same functionality of the global pointer, synchronization structure, and remote function invocation without any language modification and it does not require a special compiler on either the homogeneous or heterogeneous computers environments.

7. Concluding Remarks

In this paper, after presenting an overview of the MPC++ parallel programming language, the MPC++ new runtime system for heterogeneous computing has been designed, implemented, and evaluated. If an MPC++ program contains remote function invocations whose arguments are only basic data types, the program runs in both the homogeneous and heterogeneous environment without any changes. If a remote function invocation contains a data structure, the user needs to define its marshalling/unmarshalling functions in the current implementation. In order to avoid such user-level definitions, we are now developing a generator for those functions using the MPC++ metalevel architecture[6].

The preliminary evaluation results show that some improvement on the the `_unmarshalName` function is needed. There are two methods. One is that the hash key is pre-calculated and stored in the name table so that the sender is passed to the receiver. The `_unmarshallName` on the receiver does not involve the hash generation. Another improvement method is that all addresses and symbols information on remote processors are kept in all processors

so that the remote address of a function is resolved on the sender. This does not require any extra cost on the receiver side. However, the table size is N times larger, where the N is the number of different processor types.

In terms of performance heterogeneity, large size data conversion should be performed on a faster processor instead of converting it on the sender. To realize this, the host table must contain performance information so that placement of the data conversion operation is decided at the runtime.

The MPC++ major application, so far, is a global operating system called SCore-D[4, 3] which enables a multi-user environment on clusters of computers. We are currently porting the SCore-D to the heterogeneous environment.

The authors believe that this paper contributes a template technique to solve issues on a runtime system of the heterogeneous environment, i.e., data conversion, marshalling/unmarshalling arguments, and function address resolution. Though the MPC++ assumes the SPMD programming style, the technique is so general that it is applicable to other parallel and distributed C++ language systems.

References

[1] http://www.globus.org/.

[2] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine. *HPC++ and the HPC++ Lib Toolkit.* http://www.extreme.indiana.edu/hpc++/docs/.

[3] A. Hori, H. Tezuka, and Y. Ishikawa. Highly Efficient Gang Scheduling Implementation. In *SC'98*, November 1998.

[4] A. Hori, H. Tezuka, F. O'Carroll, and Y. Ishikawa. Overhead Analysis of Preemptive Gang Scheduling. In D. G. Feitelson and L. Rudolph, editors, *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, April 1998.

[5] Y. Ishikawa. Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document –. Technical Report TR–96012, RWC, September 1996. *This technial report is obtained via http://www.rwcp.or.jp/lab/pdslab/mpc++/mpc++.html.*

[6] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Reflection '96*, pages 141–154, 1996.

[7] C. Kesselman. CC++. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, pages 91–130. MIT Press, 1996.

[8] W. G. O'Farrell, F. C. Eigler, S. D. Pullara, and G. V. Wilson. ABC++. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, pages 1–42. MIT Press, 1996.

Are CORBA Services Ready to Support Resource Management Middleware for Heterogeneous Computing? *

Alpay Duman, Debra Hensgen, David St. John, and Taylor Kidd

Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

Abstract

The goal of this paper is to report our findings as to which CORBA services are ready to support distributed system software in a heterogeneous environment. In particular, we implemented intercommunication between components in our Management System for Heterogeneous Networks (MSHN[1]) using four different CORBA mechanisms: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), Untyped Event Services, and Typed Event Services. MSHN's goals are to manage dynamically changing sets of heterogeneous adaptive applications in a heterogeneous environment. We found these mechanisms at various stages of maturity, resulting in some being less useful than others. In addition, we found that the overhead added by CORBA varied from a low of 10.6 milliseconds per service request to a high of 279.1 milliseconds per service request on workstations connected via 100 Mbits/sec Ethernet. We therefore conclude that using CORBA not only substantially decreases the amount of time required to implement distributed system software, but it need not degrade performance.

1 Introduction

This paper describes the experiences we had using CORBA mechanisms to implement intercommunication in MSHN. MSHN's goal is to support the execution of multiple, disparate, adaptive applications[2] in a dynamic, distributed heterogeneous environment. To accomplish this goal, MSHN consists of multiple, distinct, and eventually replicated distributed components that themselves execute in a heterogeneous environment. These components have widely varying functionality, come in and out of existence, and communicate across heterogeneous networks. In addition to executing on different types of platforms, these components are also likely to be written in different programming languages. We can, of course, at the expense of a great deal of programmer's time, implement specialized naming services to locate the appropriate component at run-time, and specialized communication mechanisms to enable communication between the heterogeneous platforms upon which the components run. Alternatively, we can use a general tool, such as the Common Object Request Broker Architecture (CORBA), to achieve the same functionality while reducing our development time. Experience with generalized systems, such as CORBA, has revealed that the reduction in development time costs come at the expense of run-time performance, which can be critical in real-time applications. This research, therefore, investigates the utility and overhead of communication mechanisms, which are implemented according to the CORBA 2.2 specification, to support MSHN's inter-component communication.

We note to the reader that our interest lies in the CORBA mechanisms that support the development of (possibly real-time) resource management environments. This is a very specific realm where system overheads can have a significant impact on performance. We do not explore the many and varied capabilities of CORBA for the supporting of other environments, such as that of distributed general database services and video streaming. Our interest in CORBA is primarily as a tool to reduce the time/programming investment needed to implement our resource management system middleware. As the services and mechanisms provided by the CORBA 2.2 specification, particularly Static and Dynamic Invocation, and the Event Services, hold great promise in this regard, we performed the series of studies detailed in this paper.

CORBA specifies a standard to permit different pro-

*This research was supported by DARPA under contract number E583. Additional support was provided by the Naval Postgraduate School and the Institute for Joint Warfare Analysis.

[1] Pronounced "mission"

[2] This paper focuses on the use of CORBA mechanisms to support the components of MSHN, not the applications that MSHN itself supports. For more details concerning applications, please see the references or contact the the authors directly.

0-7695-0107-9/99 $10.00 © 1999 IEEE

grams, executing on different computers, to request services from one another. CORBA's Naming Service and Object Request Brokers (ORBs) aid clients in locating appropriate servers. CORBA's static invocation enables a CORBA client to make a request of a server that is identified prior to compile time. It provides both reliable synchronous semantics and unreliable asynchronous semantics. In contrast, CORBA's dynamic invocation enables the client to locate a server that may not be known until run-time, and provides reliable synchronous and asynchronous semantics, as well as unreliable asynchronous semantics. CORBA's event services allow processes on one machine to place event notifications intended for processes on other machines into event queues so that the notifications can later be delivered to the serving processes. This service facilitates multicast. This paper will not cover CORBA in detail, but there are many other good references on the subject [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

The paper is organized as follows. We first briefly describe MSHN, concentrating on the type of intercommunication that is required by its components. A more complete description of MSHN can be found elsewhere [12]. Alternate designs for facilitating communication within MSHN itself and the implementation of these designs are presented. These designs are based upon, respectively, static invocation, dynamic invocation, untyped event service and typed event service. In this section, we also provide a qualitative assessment detailing the problems that we encountered while attempting to use these mechanisms within MSHN. In a subsequent section, we describe our experiments for evaluating these mechanisms within MSHN and present a quantitative analysis of each of the mechanisms. Finally, we summarize our findings.

2 The Management System for Heterogeneous Networks (MSHN)

In the Heterogeneous Processing Laboratory at the Naval Postgraduate School, we are designing, implementing, and testing a resource management system called the Management System for Heterogeneous Networks (MSHN). MSHN is designed as a general experimental platform for investigating issues relating to the design and construction of future resource management systems operating in heterogeneous environments. Though MSHN is used to explore a large number of such issues, our present research focuses on finding and developing (1) mechanisms for supporting adaptive applications, (2) mechanisms for supporting the satisfaction of user and system defined Quality of Service (QoS) requirements, and (3) mechanisms for acquiring and usefully aggregating measurements of both

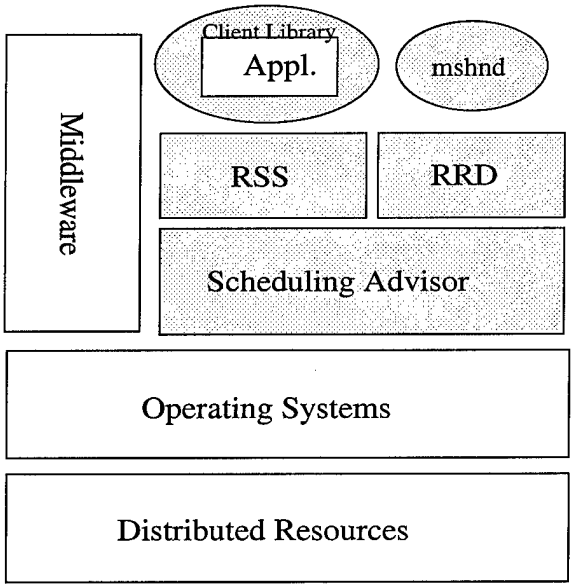


Figure 1: MSHN Conceptual Architecture

general resource availability and the resource usage of individual tasks. A thorough and complete description of MSHN can be found in Hensgen [12].

MSHN's architecture consists of multiple instantiations of each of the components enumerated below:

- a Client Library (one for each executing application to be managed by MSHN),
- a Scheduling Advisor (hierarchically replicated),
- a Resource Requirement Database (hierarchically replicated),
- a Resource Status Server (hierarchically replicated), and
- a MSHN Daemon (when needed).

Figure 1, the MSHN Conceptual Architecture, shows all of the MSHN components (shaded) as **translucent layers** executing on distributed platforms. A translucent layer is one that can be bypassed by layers that are above or below it. For example, the MSHN Daemon (mshnd) can interact directly with the operating systems layer, bypassing the Resource Status Server, the Resource Requirement Database and the Scheduling Advisor. In the environment that MSHN supports, both MSHN and non-MSHN applications may be executing at any given time. Figure 2 illustrates how these components, along with various MSHN and non-MSHN applications, might actually be distributed among different heterogeneous machines.

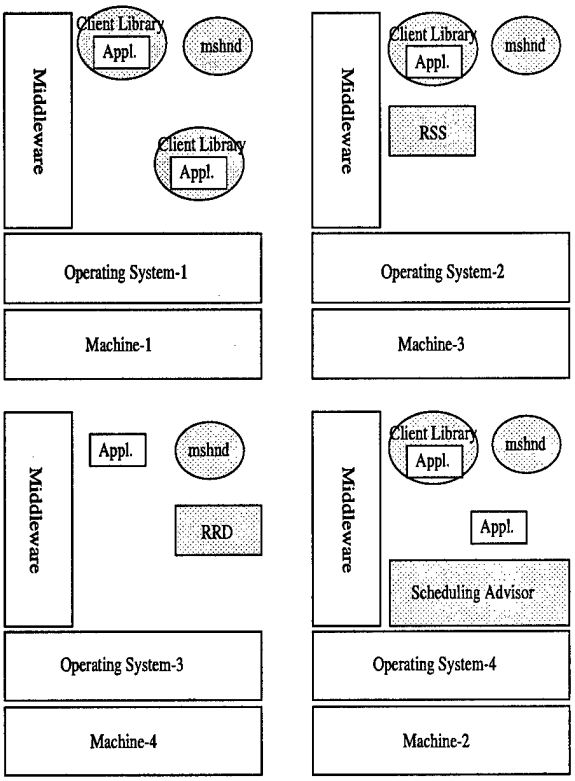


Figure 2: Example MSHN Physical Instantiation

This research investigates how communication between the components can be facilitated. As such, the MSHN description in the remainder of this section emphasizes that communication.

Figure 3, MSHN's Software Architecture, illustrates all of the interactions between the components. MSHN has a peer-to-peer architecture[3].

We now present two- and three-tier views to give a clear understanding of the interactions between the components. Generally, many applications, each linked with the MSHN Client Library, will be running at any given time. They will need to communicate with a Scheduling Advisor (SA) to request the appropriate resources needed to start new processes. They may also communicate with a MSHN Daemon when receiving their recommended schedule. Additionally, their Client Libraries update the Resource Requirement Database (RRD) and the Resource Status Server (RSS) with the expected resource requirements of the applications and current resource availability within the MSHN system.

[3]When callbacks are used the client and the server have a peer-to-peer relationship. In distributed systems, callbacks are useful as a mechanism for performing asynchronous communication. Callbacks transmit event notifications without blocking the event originator. Callbacks flow from the servers towards the clients.

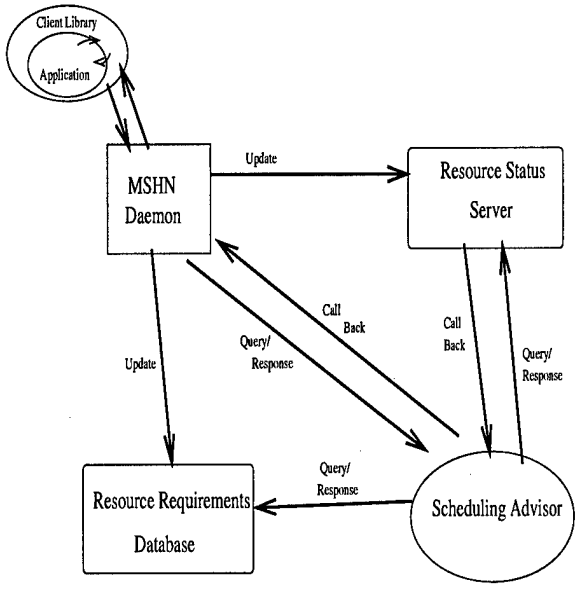


Figure 3: MSHN's Software Architecture

Figure 4 illustrates this updating interaction as a two-tiered client/server architecture. The arrows labeled "1" designate the Resource Requirements Database update path, and those labeled "2," the Resource Status Server update path. The update frequency of the Resource Status Server is expected to be high so that it, in turn, can supply the Scheduling Advisor with accurate and current information.

We anticipate that the frequency of the updates will load down the network, and cause a considerable processing load on the Resource Status Server and the Resource Requirement Database. To avoid these loads, MSHN's design includes proxy Resource Status Servers and Resource Requirement Databases that will come in and out of existence as required to minimize the number of updates. These proxies will filter gathered information and update the hierarchical Resource Status Server and the hierarchical Resource Requirement Database when necessary.

In one view, the Scheduling Advisor functionally resides between the information needed to create a schedule (the Resource Status Server and the Resource Requirement Database) and the requesters of schedules (applications linked with the Client Library). This indicates that there will be a high communication rate to and from the Scheduling Advisor. We can therefore also view MSHN as having three tiers, where the Scheduling Advisor is the second tier, and the Resource Status Server and the Resource Requirement Database are in the third tier (see Figure 5). When the Client Library (first tier) contacts the Scheduling Advisor for

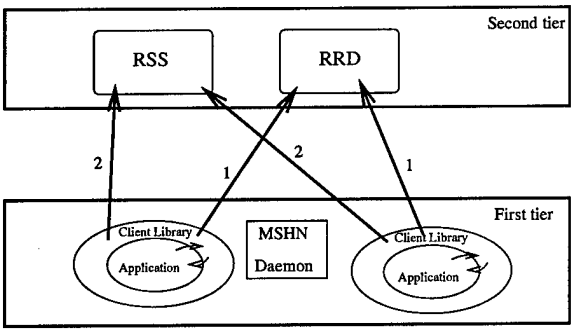


Figure 4: Two-tiered Architectural View of MSHN Architecture

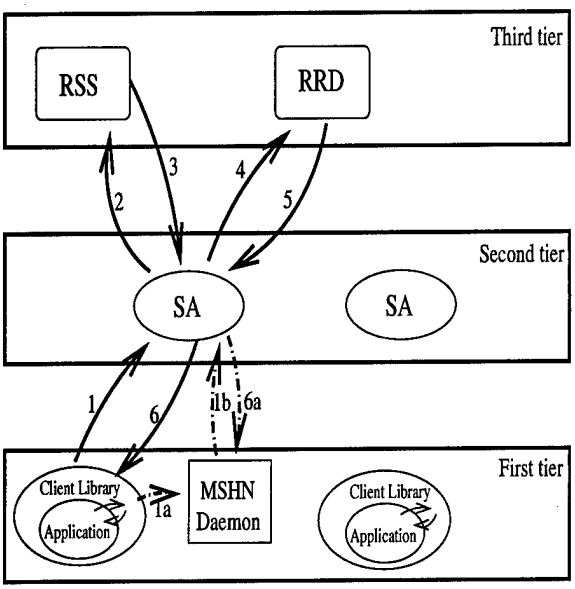


Figure 5: Three-tiered View of MSHN

a schedule, either directly or via the MSHN Daemon (the arrows labeled "1" and "1a"), the Scheduling Advisor queries both the Resource Status Server (arrows "2" and "3"), and the Resource Requirement Database (arrows "4" and "5") before it computes its schedule and sends it to the MSHN Daemon or client library depending upon which is more appropriate (arrows "6" and "6a")

Although the Client Libraries are the initiators of many of the communication chains through the MSHN system, other chains are initiated by the Resource Status Server. For example, in the case where a violation of a deadline occurs because of a change in resource availability, the Resource Status Server will trigger the Scheduling Advisor to reschedule processes that would not otherwise meet their deadline. The Scheduling Advisor will adapt to the new situation by either changing

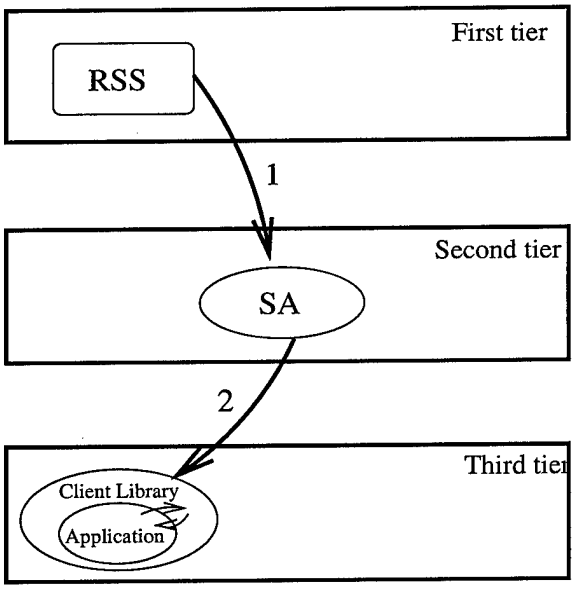


Figure 6: Alternate Three-tiered View of MSHN

the format[4] of the process or restarting it on a different resource, possibly via the MSHN Daemon. This interaction is the reverse of the previously described communication chain and can be used to define another version of a three-tiered view. (See Figure 6.)

Although we have shown several two and three tier views of MSHN, the reader should understand that these are only examples. Much larger chains will actually exist when the various components are hierarchically replicated.

3 Use of CORBA Services in MSHN and Problems Encountered

Our goal is to determine both (1) how we can best facilitate efficient communication between the components in our architecture using mechanisms from the CORBA 2.2 specification, and (2) to determine the runtime overhead of each of those mechanisms. Our justification for choosing a particular mechanism included extensibility, scalability, portability, flexibility, and efficiency.

MSHN consists of multiple, eventually replicated, distinct distributed components that execute in a heterogeneous environment. These components will have widely varying functionality, will come in and out of existence, will communicate via heterogeneous networks, and will execute on different platforms. To facilitate the interactions between MSHN's components, we identified four mechanisms from the CORBA 2.2 specifica-

[4] We use the term "format" to refer to a mechanism we have developed to support adaptive applications [13].

tion that had particular promise: the Typed Event Service, the Untyped Event Service, the Static Invocation Interface (SII), and the Dynamic Invocation Interface (DII). After settling on these four mechanisms, we implemented a prototype of MSHN's communication infrastructure using each of them. First we describe how the MSHN architecture would benefit from the both the Typed and Untyped Event Service, the Static Invocation Interface (SII), and the Dynamic Invocation Interface (DII). Then we discuss how we use the Naming Service within MSHN to obtain object references. In this section, since part of the objective of this paper is to make recommendations with regards to additions and improvements to the evolving CORBA specification, we describe and justify each of our designs, the problems we encountered, and the solutions to which we arrived.

3.1 Selection of a CORBA ORB

At the beginning of this research, we explored various implementations of the CORBA standard. Figures 7 and 8 present a summary of the results of that exploration[5]. Based upon various requirements, including the cost of some of the implementations, the time required to implement comparative tests, and the duration of this study, we had to limit ourselves to one CORBA implementation. We chose the implementation that seemed, at that time, to have the most mature features relevant to MSHN. Our assumption was that once such an implementation was found, other implementations would typically have similar difficulties and comparable performance. As such, we based our studies around IONA's Orbix, the implementation that best fit this requirement.

3.2 Event Service

Event Service allows multiple suppliers and multiple consumers to deliver and receive notifications for a set of events. An Event Channel transparently permits (1) suppliers to send notifications of events and (2) consumers to receive these notifications, all without knowledge of the existence of one another. Hence, the Event Service will support the transparent replication of MSHN system components for reliability and dependability. Event Service will enable Client Libraries, linked with different concurrent applications, to communicate with other MSHN components seamlessly. Finally, Event Service supports a standard Application Programming Interface (API) (e.g., for the Push-Push Model, a single operation `push()` taking a variable of type **any** as a parameter) which eases the development of MSHN system components.

Though there are four models for Event Service, there were only two available in relatively robust industrial implementations when we performed our experiments: the Push-Push Model and the Pull-Pull Model [14]. Using the Pull-Pull Model creates an additional load on the consumers. Because our servers, the consumers in this case, must minimize their use of computing resources even when there is no event to be delivered on the Event Channel, we chose to use only the Push-Push Model.

3.2.1 Using Event Service in MSHN

Figure 9 illustrates the use of Event Service to organize communication in the MSHN architecture. In this approach, the components of MSHN must register themselves as both a consumer and a supplier to the Event Channel. The Event Channel acts as the glue between all of the components and delivers notifications to each of them.

3.2.2 Problems with Initial Approach

Although this approach helps to organize MSHN's communication, providing transparent reliability and scalability, some problems can be seen involving both performance and the CORBA 2.2 specification. Some of the problems with this approach are identical to the problems identified by Schmidt and Vinoski in the analysis of their stock market application [11]. We first summarize their findings in the first two items below, Loss of Events in the System, and Problems with the Untyped Event Service. Then we enumerate additional problems that are particular to using CORBA within the MSHN architecture. Lastly, we look at how to implement a component that is both a supplier and a consumer.

Loss of Events in the System. Event Service guarantees delivery of notifications to all registered consumers as long as the Event Service process does not fail[6]. However, in the Event Service specification, persistency of events in the Event Channel is not required. Therefore, if an Event Service process does fail, undelivered notifications in the system may be lost.

The loss of notifications is fatal for MSHN because we are creating an environment for mission-critical applications. The obvious solution to this problem is to

[5]The capabilities of the various implementations of CORBA evolve very quickly. The content of these figures present the state of some of the implementations at the time this research was performed. As the capabilities of most CORBA implementations can quickly change, the reader is recommended to do his own similar exploration.

[6]Although there are many definitions of failure, we specifically mean that if the Event Service does not fail, then all consumers receive the correct value. This agrees with Lamport's definition of failure [15].

Vendor	Naming	Life Cycle	Event	Trading	Identity	Relationships
Expersoft	yes		yes			yes
Sun	yes	yes	yes		yes	yes
IONA	yes		yes	yes		
Visigenic	yes		yes			
BEA						
ICL			yes			
HP	yes	yes	yes	yes		
IBM	yes	yes	yes		yes	
Chorus						
OOT	yes			yes		
Electra	yes	yes	yes			
Xerox						
BBN	yes	yes				

Figure 7: Available Services

Vendor	Concurrency	Externalization	Persistency	Transactions	Security
Expersoft					
Sun					
IONA				yes	
Visigenic				yes	
BEA					yes
ICL				yes	yes
HP				yes	
IBM	yes	yes	yes	yes	
Chorus					
OOT		yes			
Electra					
Xerox					
BBN			yes		

Figure 8: Available Services (Continued)

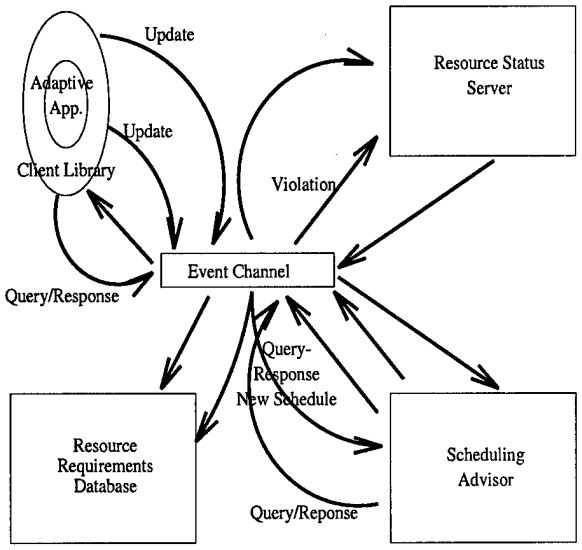


Figure 9: Using Event Service in MSHN

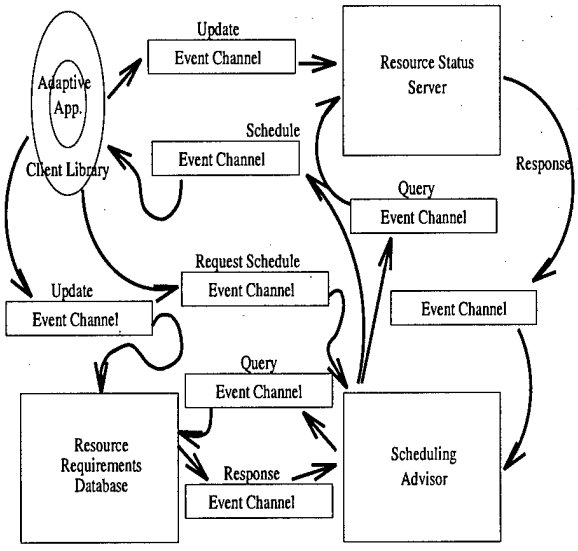


Figure 10: Using UntypedEvent Service

redefine the Event Service specification to include persistency for the undelivered notifications in the Event Channel. The OMG has been defining this requirement in the Notification Service specification [7]. However, no vendors had implemented this new specification at the time of this research.

Problems with Untyped Event Service. The Untyped Event Service does not specify any way to filter notifications. Therefore when using this service, all notifications are received by all registered consumers.

Passing all of these notifications in MSHN, many of which will be discarded by any particular consumer, through the network will increase the network load between the Event Channel and the consumer. Additionally, the consumers must filter events and convert the parameters that have type **any** to the type that is expected. In this case, there is an additional and unwanted load on the consumers to process all the events received. Finally, when more suppliers, in particular more applications, register with the Untyped Event Channel, more events will be generated in the system. Since the Untyped Event Channel delivers each event to all of the registered consumers and the consumers will filter all the events, the network load and consumer load will increase rapidly.

To handle this problem, we can use Typed Event Channels which filter the notifications according to their type. With this solution, the consumers receive only the notifications for which they register, decreasing the network traffic. In this solution, one Event Channel processes all of the notifications and delivers them only to the corresponding consumers. This also lightens the loads on the consumers because they avoid having to examine and discard events not meant for them. However, we note that it increases the computational load on the Event Channel. Later, we compare the run-time performance of Typed Event Channel to Untyped Event Channel using this approach in the MSHN architecture.

Alternatively, since we only have five different types of components in MSHN, we could use different channels for each connection between these components. In this approach, each Event Channel will only support one notification type. For example, for the Client Library - Scheduling Advisor Event Channel, we will have the Client Library as a supplier, the Scheduling Advisor as a consumer, and the possible client scheduling requests as the types of the notifications. Each MSHN component may be replicated by registering the additional (identical) components to the same Event Channel. This solution is shown in Figure 10.

Obviously, some combination of these two solutions may be best. That is, the Typed Event Channel itself can become a bottleneck in the first solution. Therefore, replication of Typed Event Channels may better fit MSHN's requirements. In this paper, we focused on the careful analysis of individual solutions rather than empirically exploring the exponentially sized solution space that combining these two techniques will create.

How to implement a component that is both a supplier and a consumer in a system in order to minimize the run-time overhead. All components of MSHN are both consumers and suppliers. Also,

and perhaps particular to MSHN, when a component receives a notification, it usually becomes a supplier by generating another notification and delivering it to the appropriate Event Channel. Figure 11 shows the process of passing notifications from the Client Library to the Scheduling Advisor using the `push()` operation. It reveals how the Scheduling Advisor changes from a consumer to a supplier. In the Untyped Event Service's Push-Push Model, the supplier (here the Client Library) invokes a default `push()` operation on the Event Channel which in turn invokes a `push()` operation supplied by the developer of the consumer (here the Scheduling Advisor). In the `push()` operation that the developer supplied for the Scheduling Advisor (as a consumer), the developer of the Scheduling Advisor invokes the default push operation on the Scheduling Advisor – Resource Requirement Database (SA – RRD) Event Channel (which of course, invokes the `push()` operation supplied by the developer of the Resource Requirements Database).

The design issue here is to determine how to supply the Interoperable Object Reference (IOR) of the SA – RRD Event Channel to the `push()` operation of the Scheduling Advisor. We want to avoid using the Naming Service every time the `push()` operation (here the push operation of the Scheduling Advisor) is invoked. Instead, the developer can locate the SA–RRD Event Channel in the servant implementation. That is, the servant implementation will obtain the IOR for the SA–RRD Event Channel, stringify the IOR, and storing it in a file. The `push()` operation implementation can retrieve these IORs from their files, as needed, and deliver generated events, thereby pushing the corresponding notifications to the channel.

Therefore in the Untyped Event Service, to react to the notification (here a request for a schedule) that the consumer receives, the developer of the consumer (here the Scheduling Advisor) must override the default `push()` operation between the Event Channel and the consumer. For example, when the Scheduling Advisor receives an event from the Client Library requesting a schedule, it will generate a query notification for the Resource Requirement Database and deliver it to the SA – RRD Event Channel. In this case, the Scheduling Advisor becomes a supplier and is required to locating the SA – RRD Event Channel. To avoid locating the Event Channel to which the supplier will deliver the notification, via the Naming Service inside the `push()` operation, the developer can locate the Event Channel in the servant implementation and obtain IORs of it. Then, the servant implementation can stringify these IORs and store them in files.

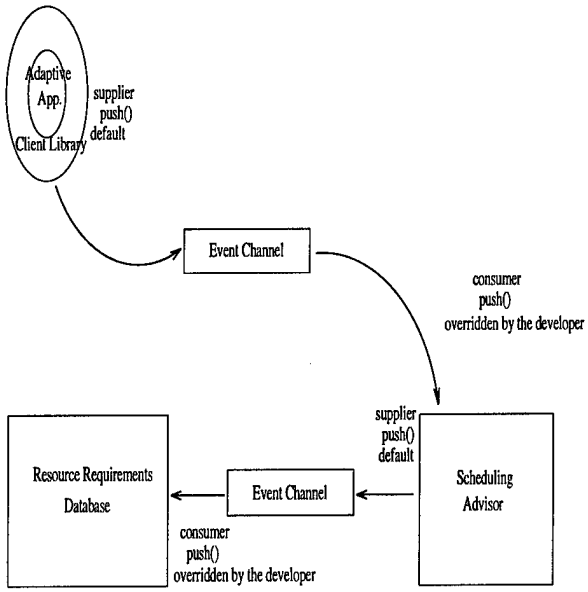


Figure 11: Using `push()` Operation

3.3 Remote Invocations

In this section, we discuss using remote invocations to coordinate the interactions of MSHN's components. Since both the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII) have similar remote invocation mechanisms, we first define the general problems encountered with both, and then enumerate any additional ones that are specific to the DII.

The same functionality described above using the Event Service can be implemented using remote invocation. The most important difference is that the replication of the components is not as easy as it is using Event Service. To support replication using remote invocation, clients must make multiple invocations rather than just the one needed in Event Service.

3.3.1 General Approach using Remote Invocation

Figure 12 shows our approach that uses remote invocations (i.e., either the Static Invocation Interface (SII) or the Dynamic Invocation Interface (DII)) to establish inter-component communication in the MSHN architecture. We chose from two communication methods available in both the SII and DII: one-way invocation and synchronous invocation, depending upon whether reliable communication is required.

When using the SII, a component requires compile-time knowledge of the Interface Description Language (IDL) interface of the target component from which it will request a service. In contrast, the same component, using the Event Service, makes its request via a

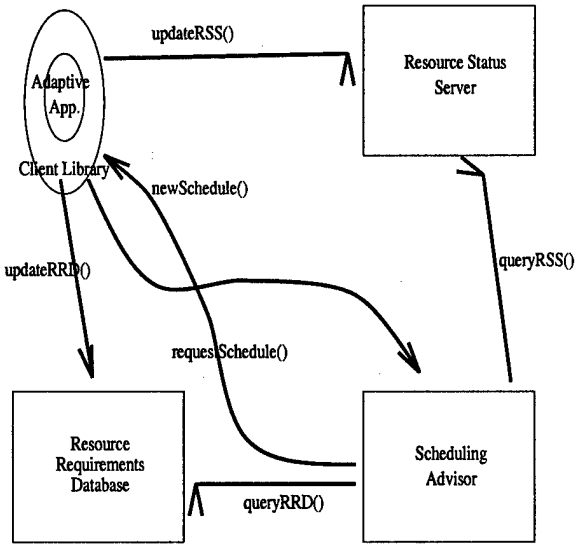


Figure 12: Using Remote Invocations in MSHN

standard API that is independent of the target component and its functionality. However, when using the DII, the components of MSHN can invoke operations on other components without requiring precompiled stubs. Thus, we may substitute different instantiations of such components without requiring a re-linking. Additionally, using the DII allows us to invoke objects using deferred synchronous invocation. Such invocation is not available from the SII within the current CORBA 2.2 specification. With deferred synchronous invocation, the clients may continue their computation instead of waiting for the results of the previously invoked operations to be delivered.

3.3.2 Problems with Using the Initial Remote Invocation Approach

We now enumerate some problems with our initial remote invocation approach.

Lack of a Standard Thread Mechanism. Our first design decision was to implement the remote invocations with threads, i.e., handling each invocation of a component using a different thread. Using threads would avoid any data synchronization problems and support fairness for each schedule request. However, the CORBA 2.2 specification does not define how the threads must be implemented. Therefore, each vendor has come up with their own solution, leading to applications that are non-portable. For example, if you use IONA's Orbix as your development environment, and IONA's Filters to implement your threads, you cannot use the same implementation on Inprise's Visibroker because Inprise's solution for handling threads uses Interceptors.

We avoided non-compliant extensions of the vendor when implementing our prototypes. Therefore, we were unable to use threads for any of our prototypes, although the usage of threads would have improved the throughput of schedule requests.

Best-Effort Semantics. One-way invocation has best-effort semantics. Thus, there is no guarantee that the requested method is actually invoked. In this mechanism, the client continues its processing immediately after initializing the request and never synchronizes with the completion of the request. Hence, one-way invocation is not a good mechanism for most of the MSHN system because it is not reliable.

However, using one-way invocations for frequent short-term updates could be cost effective in some cases in MSHN. There are two advantages to selectively using best-effort asynchronous semantics between MSHN's Client Library and Resource Status Server. First, the Client Library can continue its computation immediately without blocking. Second, we expect that the Resource Status Server will be updated very frequently. Therefore, we can afford the delay needed to get the accurate status of a resource with the next update instead of forcing the use of a more reliable transmission mechanism.

3.3.3 Problems with Our Initial Approach that are Specific to using DII

We now enumerate some problems with our initial approach that are specific to using DII.

The Additional Overhead of the DII. A straight forward DII approach requires 5-6 method invocations in order to invoke a single remote method: looking up the interface name, getting the operation identifier/parameters, and creating the request (which may also be remote). This would add a lot of overhead to run-time performance, which would be unacceptable in MSHN's architecture.

In MSHN however, we know the interface of the components, i.e., the operation identifier, the parameters and the return type, when we are developing the client applications. Thus, we can obtain the flexibility and benefits the DII's deferred synchronous invocation, without having to pay the overhead of querying the Interface Repository for the interface information. We do note that if a deferred synchronous invocation, such as Promises [16], had been specified as part of CORBA's static invocation interface, the use of DII would not be necessary in this case. We compare the performance of the SII and DII in the results section.

3.4 Using the Naming Service

We used the Naming Service to obtain object references in each of our prototypes. For the static and dynamic invocation interfaces, all components must resolve names only once, when they are instantiated, to obtain IORs via the Naming Service. References within all components, except the Client Library, are stored in files for future use as we described previously. The components do not use the Naming Service unless the IORs that they have are no longer valid. We use the exception handling mechanism in CORBA to catch non-valid IORs, and then use the Naming Service to obtain new valid ones.

To improve the run-time performance of the Event Service implementations, we registered each component with the appropriate Event Channel. We resolve the Event Channel references using the Naming Service. Then we query the Event Channels to obtain the references for the Proxy Push Suppliers, stringify them, and then store them in files. When a component receives an event, and generates another event in response to the one it received, that component reads the appropriate file to obtain the stringified reference and uses this reference to push the event to the corresponding Event Channel.

4 Quantitative Results

We described our design decisions for implementing our prototypes in the previous section. In this section, we discuss the performance results of these different prototypes. First, we describe our test bed. Then we explain our tests and enumerate their results.

4.1 Hardware and Software Used in the Test Bed

As discussed earlier at the beginning of this research, we surveyed the available implementations of CORBA to determine what services were supported. (See Figure 7 and 8.) Based upon the robustness and availability of services, particularly the Typed Event Service, we chose IONA Technologies' CORBA implementation, specifically OrbixMT2.3c, OrbixNames1.1c, OrbixEvent1.0c (Untyped Event Service) and OrbixEvent1.0b (Typed Event Service) built using the SunSparc C++ Compiler 4.1.

We ran our tests on SunSparc Station 10 hosts with 300MHz CPUs and 128 MB of RAM each, running the Solaris 2.6 operating system. The hosts were connected via a 100 Mbits/sec Ethernet LAN.

To obtain correct results in the tests utilizing the network, we used the Network Time Protocol to synchronize the system clocks of the hosts. We found that the system clock on the SunSparc 10 has a skew of approximately 3 milliseconds every 15 minutes. Therefore in order to minimize the difference between the various system clocks, we synchronized the clocks every 5 minutes and ran the tests immediately after the synchronization.

4.2 Experiments

We determined the overhead of each CORBA mechanism on a single machine, and then measured the response times over the network of the various mechanisms, that is, the total time required to service 1000 scheduling requests. This interval begins when the Client Library requests a schedule from the Scheduling Advisor and includes all processing up until the time that the Client Library receives a response. This duration includes the time spent querying the Resource Requirement Database and the Resource Status Server. At the time of this testing, we did not have a fully functional Scheduling Advisor, so we emulated its execution by having the thread that was computing a schedule pause for .5 seconds. We chose this duration based upon the average execution time of a set of 11 scheduling algorithms proposed for MSHN's repertoire by Siegel [17].

To assess the overhead of CORBA, we included one non-CORBA test. This base case consists of an application linked with all the MSHN components and executing as a single process on a single host. This non-CORBA test uses local method invocation to perform MSHN component intercommunication. In order to assess CORBA's overhead, we performed two sets of tests. In the first set, we compared this base case against test cases where we ran all the MSHN components on the same machine and had them communicate via CORBA mechanisms. In the second, we compared the latter tests against ones where the MSHN components are distributed across different machines.

With the exception of the non-CORBA base case, we ran all tests both on a single machine and over the network using different workstations to execute each of the Client Library, the Resource Status Server, the Resource Requirements Database and the Scheduling Advisor.

All single machine CORBA tests were executed using four different processes. The non-CORBA single machine tests executed completely in a single process, with all MSHN calls being implemented as ordinary C++ function calls. In implementing both static invocation and dynamic invocation for a single machine, we used synchronous semantics.

The average inter-arrival rate of schedule requests varies with the facility and time of day. Therefore, we ran all of our tests for two different circumstances. In the first, the inter-arrival rate of the requests is less than the service time, i.e., each request is completed

by the system before the next request arrives on average. The second represents the situation that exists in the middle of a burst. In this case, the inter-arrival rate of the requests is greater than the service time, i.e., some requests must be queued to be handled later. The first case is important in determining performance under normal conditions, but it is equally important for us to determine that the system neither (1) fails completely when heavily loaded, nor (2) incurs overhead that varies exponentially with the number of requests pending. Indeed, no typed event service that we have tested to date could pass the above stress tests.

Unfortunately, the system clocks had insufficient granularity to measure precisely the total time to process a single request in our non-CORBA implementation. We therefore first read the system clock. We then generate a request and await its response, repeating this 1000 times. Lastly, we read the clock again, and determine the total time (for 1000 consecutive request-response pairs). Because requests are generated consecutively, and because each request uses synchronous semantics to make the invocations, we call this set of tests, the **consecutive synchronous tests**.

To simulate the case where many requests occur within a short time frame, we generated requests every .06 seconds, on average, in our base case. For this set of tests, we used asynchronous calls within the application to start the schedule request chain in the DII and SII implementations. Event Service is meant to be used asynchronously, so there was no special programming required to implement these cases. We call this set the **bursty asynchronous** tests because during such a burst, the requests arrive faster than the expected required service time and queue up for the Scheduling Advisor.

For another of our projects, Schnaidt and Duman implemented a fully optimized version of an application using sockets and compared it to an equivalent CORBA implementation to determine CORBA's overheads when running over the network [18]. As such, we did not implement such a socket implementation of MSHN. In the following paragraphs, we draw some conclusions based both on the Schnaidt-Duman experiments and those reported here.

4.3 Results

We summarize our quantitative results in Figure 13. The times shown are the actual execution times, in seconds, for 1000 requests. We have included a scheduling time of .5 seconds per request and have not simulated the execution time of the application.

In order to fully understand these results, we must first explain some anomalies that we observed in the Unix calls we used to emulate the Scheduling Advisor

Config.	Communication Mechanism	Local	Network
Consec. Synch.	Non-CORBA	500.1	N/A
	SII	511.4	520.0
	DII	530.1	530.4
	Untyped Event	607.4	593.9
	Typed Event	580.5	779.2
Bursty Asynch.	Non-CORBA	500.1	N/A
	SII	510.8	510.8
	DII	521.2	520.2
	Untyped Event	592.8	564.4
	Typed Event (for 100 requests)	64.7	63.6

Figure 13: Results of the Generic Experiments for 1000 Requests

(`select()`) and the request generation inter-arrival rate (`ualarm()`). The average of the actual `select()` times was 125 microseconds more than the requested .5 seconds. We also observed an average of 10 milliseconds error for the `ualarm()` requests of 60 milliseconds.

As expected, there is significant overhead in using CORBA for communication, and therefore across more than one address space, as compared to local invocations within a single address space. In our earlier project, we noted similar results as well as substantial overhead when an optimized non-CORBA local socket implementation was compared to a local CORBA implementation [18]. The efficiency of the socket implementation on a single machine is due to its use of shared memory. However, even if a CORBA implementation used shared memory, comparable performance would not be obtained. Unfortunately, the CORBA specification requires all parameters of a request to be converted to an external, machine independent data representation, even if the target object resides on the same machine. Also, in that earlier project, we noted that a networked CORBA implementation, which required less than 5% of the time to implement as compared to the socket implementation, had only 20% more runtime overhead. Since our results are comparable here, and because we did not implement a highly optimized MSHN socket implementation, we will limit the remainder of our remarks to comparing the performance of various CORBA implementations of MSHN.

Static invocation is generally the fastest intercommunication mechanism available in CORBA [1]. Even though dynamic invocation is generally much slower, we see that the performance of dynamic invocation, when we know the interfaces at development time, is close

Communication Mechanism	Added Overhead
SII	10.5
DII	20.0
Untyped Event	64.2
Typed Event	13.5

Figure 14: Added Overhead for Bursty Asynchronous Test Case over the Network

to that of static invocation. However, we note that the most efficient implementation would likely be available from a deferred synchronous Static Invocation Interface. We recommend that such semantics, similar to those in Promises [16], be considered for adoption into the CORBA specification.

The comparison between the consecutive synchronous and bursty asynchronous tests seems surprising at first glance. One would normally expect that a system loaded with bursty requests would not perform better than an unloaded system. To understand the reason for this performance improvement, we must further elaborate on the client application's use of the Naming Service. In the consecutive case, the Client Library obtains the reference of the Scheduling Advisor from the Naming Service immediately prior to making each request. However, in the bursty asynchronous case, the Client Library obtains all of the references asynchronously. Thus in the bursty asynchronous case, obtaining these references overlaps with the actual computation. Unfortunately, we will only expect to see this improvement in the actual MSHN implementation if the Scheduling Advisor is executing on a dual processor machine. In our experiments, the emulated Scheduling Advisor is actually blocked while the Naming Service is resolving addresses.

In the 4-machine network tests, the number of context switches required between MSHN's components and the Object Request Broker is substantially reduced. Multiple components actually execute simultaneously, and thus run-times were smaller.

As seen in Figure 13, the Untyped Event Service adds more overhead than either static or dynamic invocation because the Event Service process is the bottleneck in the system. Of course in an overall evaluation, this additional overhead must be balanced against the reduced cost with which information can be delivered to replicated system components.

In addition to the tests described above, we replicated the Untyped Event Service to see whether any speedup could be obtained by distributing the load of the Event Service process. First we created two Event Service processes, one on the same host as the application and the other on the same host as the Scheduling Advisor, in an attempt to achieve some speed up. This approach performed worse than the single Event Service process. Upon analysis, we determined that it introduced unnecessary network communication and placed the Event Service processes on the busiest hosts. Then we moved the Event Service processes to the same hosts as the Resource Requirements Database and the Resource Status Server. Figure 15 shows the speedup we observed with this configuration. We also ran tests using four distributed Event Service processes. Unfortunately, probably because of the excessive amount of communication, this approach performed no better than using a single Event Service process.

In MSHN's Typed Event Service implementation, all of the communication passes through a single process. The CORBA implementations that we used[7] failed in this bursty asynchronous case. In Figure 13, we include the time required to process 100 requests for the bursty asynchronous case. Since the current implementations of Typed Event Service do not allow replication, we could not run a replicated test with the Typed Event Service as we did with the Untyped Event Service. Hence, we believe that the Typed Event Service is not ready to be used in middleware to support heterogeneous distributed computing.

5 Conclusions

In this paper, we described our experiences using mechanisms of the CORBA 2.2 specification to facilitate communication in a resource management system that is both designed to manage distributed heterogeneous applications, and is itself distributed and heterogeneous. In our qualitative assessment of CORBA 2.2, we found several minor problems and recommended the addition of deferred asynchronous semantics to CORBA's Static Invocation Interface. We found that both CORBA's static invocation and dynamic invocation, when used solely to obtain asynchronous semantics, were efficient enough to support distributed heterogeneous resource management systems. We found that substantial work is needed to provide implementations of Typed Event Services that can handle the loads placed on them when requests occur in a bursty fashion. We also determined that while Untyped Event Services add substantial overhead as compared to static invocation, they may still be desirable in the case where multicast of requests is desired, particularly if they are replicated and themselves wisely allocated to machines in the system. In summary, many of the

[7] Typed Event Service is new in the CORBA 2.2 specification and not many CORBA products have this service available as yet.

	Replication Mechanism	All	SA and Client Hosts	RRD and RSS Hosts
Bursty Asynch.	Two Event Pro.	N/A	574.38	561.44
	Four Event Pro.	560.98	N/A	N/A
Consec. Synch.	Two Event Pro.	N/A	599.05	593.82
	Four Event Pro.	593.82	N/A	N/A

Figure 15: Results of the Untyped Event Service Special Cases

existing CORBA services can be quite useful in implementing resource management systems for heterogeneous computing, and other CORBA services hold substantial promise for the future.

6 Acknowledgements

The authors would like to thank Ted Lewis for his suggestions during this research as well as for sharing his broad understanding of the motivation behind the CORBA specifications.

References

[1] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. John Wiley, New York, 1997.

[2] Robert Orfali, Dan Harkey, and Jeri Edwards. *Distributed Objects*. John Wiley, New York, 1997.

[3] Sean Baker. *CORBA Distributed Objects Using Orbix*. Addison Wesley Longman Limited, Essex, 1997.

[4] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide*. John Wiley, New York, 1997.

[5] IONA Technologies PLC. *Orbix Programmer's Reference Manual*, October 1997.

[6] IONA Technologies PLC. *Orbix Programmer's Guide*, October 1997.

[7] Object Management Group. *CORBA 2.2 Specification*, February 1998.

[8] Object Management Group. *Naming Service Specification*, November 1997.

[9] Object Management Group. *Event Service Specification*, November 1997.

[10] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[11] Douglas C. Schmidt and Steve Vinoski. Overcoming drawbacks in the OMG events service (column 10). *SIGS C++ Report Magazine*, June 1997.

[12] Debra A. Hensgen, Taylor Kidd, Matthew C. Schnaidt, David St. John, Howard Jay Siegel, Tracy D. Braun, Muthucumaru Maheswaran, Shoukat Ali, Jong-Kook Kim, Cynthia Irvine, Tim Levin, Roger Wright, Richard F. Freund, Michael Godfrey, Alpay Duman, Paul Carff, Shirley Kidd, Viktor Prasanna, Prashanth Bhat, and Ammar Alhussaini. An overview of MSHN: A Management System for Heterogeneous Networks. In *8th. IEEE Workshop on Heterogeneous Computing Systems (HCW'99)*, San Juan, Puerto Rico, April 1999. IEEE, IEEE. invited.

[13] John Kresho. Quality network load information improves performance of adaptive applications. Master's thesis, Naval Postgraduate School, September 1997.

[14] IONA Technologies PLC. *OrbixEvents Programmer's Guide*, December 1997.

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[16] B. Liskov and L Shirira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *Proceedings SIGPLAN'88 Conference Programming Design and Implementation*, 1988.

[17] Tracy D. Braun, Muthucumaru Maheswaran, Howard Jay Siegel, Noah Beck, Ladislau Boloni, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, and Bin Yao. A taxomony for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. *Proceedings of the Workshop on Advances in Parallel and Distributed Systems (ADAPS)*, 1998.

[18] Matthew Schnaidt and Alpay Duman. A comparison of Unix sockets and CORBA in a distributed communication intensive application. Technical Report 3, Naval Postgraduate School, 1998.

Alpay Duman is LTJG in Turkish Navy. He graduated from the Turkish Naval Academy getting his BS in Operations Research with honor degree. He received his Ms.Cs. degree in the area of Systems Design and Architecture from the Naval Postgraduate School. He investigated the use and runtime overhead of CORBA

in DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) with Dr. Debra Hensgen and Dr. Ted Lewis. He is currently a systems engineer at Turkish Navy Software Development Center working on a CORBA based communication infrastructure for Command Control Systems. His area of interest are fault tolerant, real-time, CORBA based distributed systems.

Debra Hensgen is an Associate Professor in the Computer Science Department at The Naval Postgraduate School. She received her PhD in the area of Distributed Operating Systems from the University of Kentucky. She is currently a Principal Investigator of the DARPA-sponsored Management System for Heterogeneous Networks QUORUM project (MSHN) and a co-investigator of the DARPA-sponsored Server and Active Agent Management (SAAM) Next Generation Internet project. Her areas of interest include active modeling in resource management systems, network rerouting to preserve quality of service guarantees, visualization tools for performance debugging of parallel and distributed systems, and methods for aggregating sensor information. She has published numerous papers concerning her contributions to the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems

David St. John is the head of staff at the Heterogeneous Network and Computing Laboratory. He plays a strong role in development of the MSHN prototype and in directing students' research at the Naval Postgraduate School. He has over six years experience in object-oriented software development primarily for process control, sensor collection, and Internet transaction processing systems. He is a member of IEEE and IEEE Computer Society. He was a recipient of the Chancellor's Fellowship and an MS degree in Engineering from the University of California, Irvine in 1994. He received his BS degree in Mechanical Engineering from the University of Florida in 1992.

Taylor Kidd obtained his Ph.D. from the University of California at San Diego in 1991. He is an active researcher in theoretical and applied distributed computing. As part of the SmartNet Team at NRaD he led the SmartNet Research Team. Recently he joined Debra Hensgen as co-director of the Heterogeneous Network and Computing Laboratory. While part of the SmartNet Team, he instigated a number of important advances to the SmartNet Scheduling Framework, including enhancing compute characteristic learning by using Student-T techniques, performing experiments and simulations exploring the performance of different RMS's in homogeneous and heterogeneous environments, and fundamentally changing the way SmartNet learns and schedules by recognizing the basic predictable uncertainty of job run times. He has served on the program committees of several conferences and has worked as an acting subject area editor for JPDC. Most recently, he is working under DARPA's Quorum program as a co-PI for MSHN and as a co- investigator on the SAAM Project.

Session III

Modeling and Analysis

Chair

Steve Chapin
University of Virginia at Charlottesville

Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment

Michael A. Iverson, Füsun Özgüner, and Lee C. Potter
Department of Electrical Engineering
The Ohio State University
2015 Neil Avenue, Columbus, OH 43210
{iversonm,ozguner,potter}@ee.eng.ohio-state.edu

Abstract

In this paper, a method for estimating task execution times is presented, in order to facilitate dynamic scheduling in a heterogeneous metacomputing environment. Execution time is treated as a random variable and is statistically estimated from past observations. This method predicts the execution time as a function of several parameters of the input data, and does not require any direct information about the algorithms used by the tasks or the architecture of the machines. Techniques based upon the concept of analytic benchmarking/code profiling [7] are used to accurately determine the performance differences between machines, allowing observations to be shared between machines. Experimental results using real data are presented.

Keywords: *Heterogeneous Distributed Computing, Execution Time Estimation, Nonparametric Regression, Analytic Benchmarking, Distance Matrices.*

1 Introduction

Heterogeneous metacomputing is a type of parallel computing, where a large, distributed network of heterogeneous machines is used as a single computational entity. Applications executing in this environment consist of a set of coarse-grained, precedence-constrained tasks, where the precedence structure can be represented using a directed acyclic graph (DAG). The performance of an application in this environment is largely determined by the manner in which these tasks are assigned to the machines; the construction of such and assignment is called the matching and scheduling problem. Matching and scheduling algorithms need to know the execution time of each task on each machine to perform well, and most matching and scheduling algorithms for DAGs in the literature assume that the execution time of a given task is a known quantity. However, the execution time of a task on a given machine depends upon many factors, including the problem size and the input data, and is not trivial to determine *a priori*. In a heterogeneous environment, the wide variety of machine architectures further complicates the process of determining the execution time, since the execution time is also machine dependent. Methods are clearly needed which can accurately predict the execution time of a task on a variety of machines as a function of the features of the data set. This problem is called the execution time estimation problem.

In the literature, there are three major classes of solutions to the execution time estimation problem: code analysis [18], analytic benchmarking/code profiling [7, 11, 12, 16, 20, 22, 23] and statistical prediction [3, 10, 13]. In code analysis, an execution time estimate is found through analysis of the source code of the task. A given code analysis technique is typically limited to a specific code type or a limited class of architectures. Thus, these methods are not very applicable to a broad definition of heterogeneous computing, and will not be examined here.

A class of methods which are more useful in a heterogeneous metacomputing environment is analytic benchmarking/code profiling. Analytic benchmarking/code profiling was first presented by Freund [7], and has been extended by Pease et al. [16], Yang et al. [22, 23], Khokhar et al. [11, 12], and Siegel [20]. Analytic benchmarking defines a number of primitive code types. On each machine, benchmarks are obtained which determine the performance of the machine for each code type. Code profiling attempts to determine the composition of a task, in terms of the same code types. The analytic benchmarking data and the code profiling data are then combined to produce an execution time estimate. Analytic benchmarking/code profiling has two

0-7695-0107-9/99 $10.00

disadvantages. First, it lacks a proven mechanism for producing an execution time estimate from the benchmarking and profiling data over a wide range of algorithms and architectures. Second, it cannot easily compensate for variations in the input data set. However, analytic benchmarking is a powerful comparative tool in that it can determine the relative performance differences between machines.

The third class of execution time estimation algorithms, statistical prediction algorithms, make predictions using past observations. A set of past observations is kept for each machine, which are used to make new execution time predictions. The matching and scheduling algorithm uses these predictions (and other information) to choose a machine to execute the task. While the task executes on the chosen machine, the execution time is measured, and this measurement is subsequently added to the set of previous observations. Thus, as the number of observations increases, the estimates produced by a statistical algorithm will improve. Statistical prediction algorithms have been presented by Iverson et al. [10], Kidd et al. [13], and Devarakonda and Iyer [3]. Statistical methods have the advantages that they are able to compensate for parameters of the input data (such as the problem size) and do not need any direct knowledge of the internal design of the algorithm or the machine. However, statistical techniques lack an intrinsic method of sharing observations between machines. By allowing observations to be shared between machines, the execution time estimate on a machine with few observations can be improved by using observations from machines with similar performance characteristics.

Given the advantages and disadvantages of both analytic benchmarking/code profiling and statistical methods, this paper presents a hybrid method, which uses analytic benchmarking techniques to create a unified set of observations describing both the input data features and the machine capabilities. This unified observation space is then used by a statistical method to produce execution time estimates. In this paper, as in much of the DAG scheduling literature, each task is assumed to have exclusive use of the machine on which it executes. Thus, the execution time of a task is not a function of the other tasks in the system, and is only a function of the machine capabilities and input data. This method models the execution time of a task as a random variable, allowing the matching and scheduling algorithm to consider the uncertainty present in the execution time estimate. (Several papers have discussed the idea of scheduling with random quantities, including King [14], Tan and Siegel [21], Armstrong [2], Li et al. [15] and Hou and Shin [9].)

The remainder of this paper is organized as follows. First, the stochastic model of the execution time of a task is presented in Section 2. Section 3 presents the prediction algorithm which uses this model. Section 4 presents the results of simulations using real data, and conclusions are offered in Section 5.

2 Modeling the Execution Time as a Random Variable

The execution time of a task on a given machine largely depends on the size and properties of the input data set. For example, the execution time of many matrix algorithms depends upon the size of the matrix. Furthermore, if the matrix algorithm was iterative in nature, the execution time may also depend upon the condition of the matrix and the desired precision of the results. In principle, it is possible to quantify these properties of the input data set as a vector of numeric parameters $X = [x^1 x^2 \cdots x^q]$. Thus, the execution time of the task can be modeled as a function $t = m(X)$ of this parameter vector. However, in many instances, it is not computationally practical to determine all q parameters. Therefore, it is assumed that only a limited number $p \leq q$ of these parameters will be explicitly modeled. Thus, the parameter vector $X = [x^1 x^2 \cdots x^p]$ will be used to model the execution time of the task. However, the presence of unmodeled factors will cause a certain amount of error to be present in an estimate of the execution time. To compensate for this error, the execution time of a task is modeled as a random variable t. This random quantity can be represented as:

$$t = m(X) + \epsilon, \tag{1}$$

where $m(X)$ is deterministic, and ϵ is purely stochastic. In this equation, ϵ represents the unmodeled factors affecting the execution time, while $m(X)$ represents the modeled factors, and therefore depends upon a p-dimensional vector of parameters $X = [x^1 x^2 \cdots x^p]$. In essence, $m(X)$ represents the mean of t given X, while ϵ represents the zero-mean random error present in the estimate.

While the unmodeled factors which affect ϵ are unknown, it is possible to determine their effect upon the execution time indirectly by estimating properties of the random variable ϵ. Additionally, while ϵ does not directly depend upon the parameter vector $X = [x^1 x^2 \cdots x^p]$, in practice, ϵ does display some dependence on the modeled parameters, due to the fact that the modeled and unmodeled parameters may not be statistically independent. Thus, some degree of correlation may exist between these sets.

Given this model, the goal of the execution time estimation problem is to obtain estimates of $m(X)$ and ϵ

for some given parameter vector X. Before presenting the specific details of how these values are estimated, examples of how two real algorithms behave under this model will be presented to illustrate the concepts presented here. The first example shows an algorithm where the set of unmodeled parameters has a very small effect upon the execution time (i.e. ϵ is small). Figure 1 shows the execution time of the Cholesky matrix decomposition algorithm for various problem sizes on a single machine. The relative continuity of this curve shows that ϵ has a very small impact on the execution time.

The opposite case is illustrated in Figure 2. This algorithm attempts to determine if a given number is prime through the process of trial division. The execution time of this algorithm, for a given number n, is essentially a function of the smallest prime factor of the number n. However, it is not practical to compute the smallest prime factor of the number n, since the computational cost of this problem is equivalent to determining if the number is prime. However, it is possible to use the magnitude of the number as a parameter, since this value bounds the magnitude of the smallest prime factor. Thus there is a loose correlation between the magnitude of the number and the execution time, which can be seen in the figure. Thus, even in this extreme example, it is still possible to obtain some information about the execution time of the task which can be used by the matching and scheduling algorithm. In the next section, the techniques used to estimate the values of $m(X)$ and ϵ will be presented.

3 Execution Time Estimation Algorithm

The algorithm developed in this paper is presented in two sections. Section 3.1 poses the execution time estimation problem as a regression problem, and presents a *k-Nearest Neighbor (k-NN)* regression algorithm to compute estimates from a set of previous observations. For clarity of presentation, the regression algorithm is described using only the vector of parameters describing the task input data. While this algorithm can compute the execution time of a task on a single machine as a function the input data set, it lacks an intrinsic ability to share observations between dissimilar machines. To eliminate this restriction, the regression vector is augmented in Section 3.2 to include a parameterization of different machines. Thus, the execution time may be estimated using previous observations as a function of both machine type and task characteristics.

3.1 Nonparametric Regression

Given that the execution time of a task is modeled as a random variable (as in equation 1), the goal of this paper is to present methods to obtain estimates $\hat{m}(X)$ and $\hat{\epsilon}$ of $m(X)$ and ϵ for a given a parameter vector X which characterizes the input data set. This will be accomplished through the use of a set of n previous observations of the execution time $\{(t_i, X_i)\}_{i=1}^{n}$, where t_i is an observed execution time for the parameter vector X_i. The parameter vectors X_i of the n previous observations are samples in the parameter space $\mathbb{R}_p$ (p-dimensional real vectors). In statistics, this problem is called a *regression problem.* Note that, as presented in this section, each machine requires a separate set of observations. This restriction will be relaxed in Section 3.2.

There are a variety of different techniques to solve regression problems, which can be divided into two classes: *parametric* techniques and *nonparametric* techniques. In general, parametric techniques require knowledge of the functional form of $m(X)$ and ϵ. Since, in this paper, it is difficult to make any assumptions about the functional form of the model without specific knowledge of the task and the machine in question, parametric techniques are not well suited to this problem [10]. Nonparametric regression techniques (also called nonparametric estimators or smoothing techniques) are considered to be *data driven*, since the estimate depends only upon the set of previous observations, and not on any assumptions about $m(X)$ or ϵ. Therefore, nonparametric techniques will be used in this paper.

All nonparametric regression techniques compute $\hat{m}(X)$ using a variation of the equation

$$\hat{m}(X) = \frac{1}{n} \sum_{i=1}^{n} W_i(X) t_i \qquad (2)$$

where $W_i(X)$ is a weighting function, or kernel [8]. Observe that, for any given vector X, $\hat{m}(X)$ is a weighted average of the execution time values, t_i, of the n previous observations. The weight function $W_i(X)$ typically assigns higher weights to observations close to the parameter X, and lower weights to observations farther away from X. This is illustrated in Figure 3 for a scalar parameter $x = A$. In practice, many nonparametric regression techniques only include in the average points within some neighborhood of the parameter X, making the estimate $\hat{m}(X)$ a local average of the observations near the parameter vector X.

A similar technique can be used to determine the properties of ϵ. As mentioned above, ϵ is a zero-mean random variable, which can have an arbitrary probability density function (pdf). This arbitrary nature of ϵ

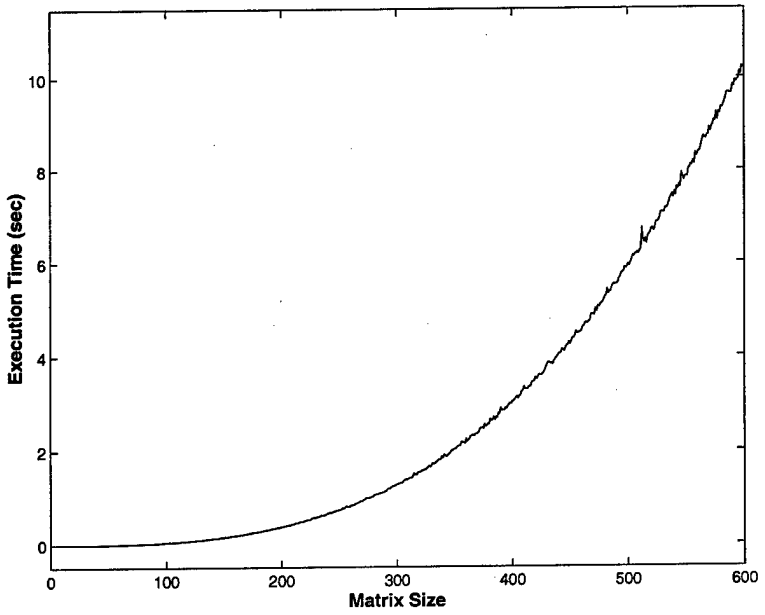


Figure 1. Execution time of the Cholesky Decomposition Algorithm.

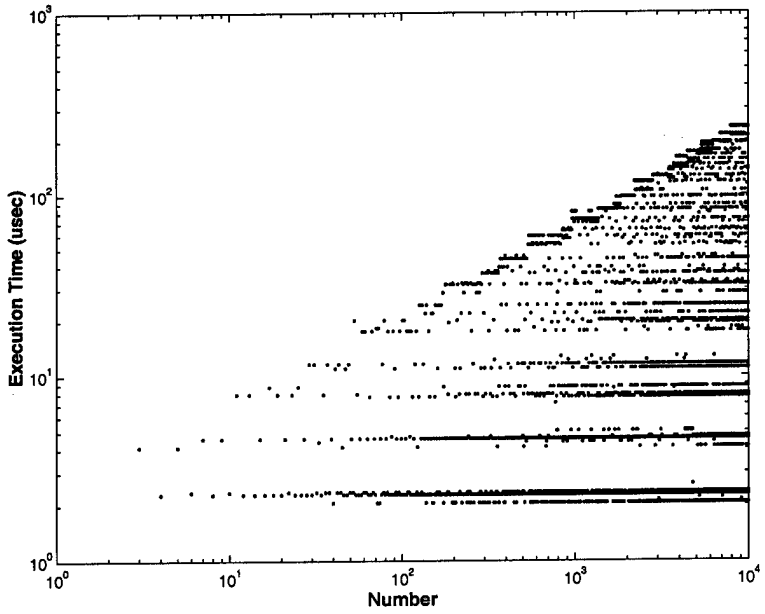


Figure 2. Execution time of the Trial Division Algorithm.

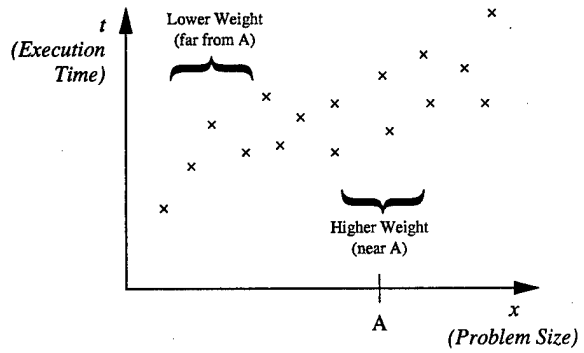


Figure 3. Assigning Weights to Observations.

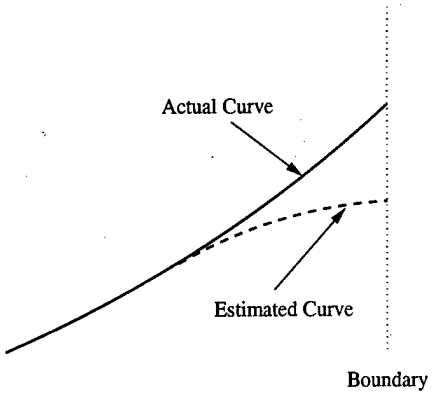


Figure 4. Effects of estimates at the boundary.

makes the estimation process difficult. To maintain simplicity, only an estimate of the variance σ^2 of ϵ will be computed in this paper. Given an estimate $\hat{m}(X)$, $\hat{\sigma}^2$ can be computed to be

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} W_i(X)(t_i - \hat{m}(X))^2. \qquad (3)$$

In this paper, the *k-Nearest Neighbor (k-NN)* algorithm is used (although, in principle, any nonparametric technique could be adapted). In *k-NN* smoothing, the estimate $\hat{m}(X)$ is constructed from the k observations with parameter vectors closest to the parameter vector X. With regard to the execution time estimation problem, there are two primary advantages of *k-NN* smoothing. First, since the estimate is always constructed from an average of k points, the method can easily adapt to sparse or dense regions in the observations. Second, the method can be implemented in a computationally efficient manner. The computational complexity of the method is $O(n(p + \log n))$, where p is the dimensionality of the parameter vector, and n is the number of past observations.

While conceptually simple, there are many factors to consider when using the *k-NN* algorithm. For example, an important issue in *k-NN* smoothing is the choice of the value k. If too many observations are included in the average, the bias $E\{\hat{m}(X) - m(X)\}$ will be too large, and the details of $\hat{m}(X)$ will be lost. On the other hand, if too few observations are averaged, the variance $E\{(\hat{m}(X) - m(X))^2\}$ of $\hat{m}(X)$ will be too large, resulting in a curve which is "noisy." Choosing k to obtain a balance between these two extremes is known as the *bias-variance tradeoff*, and is present in all nonparametric regression techniques. In general, k should grow in proportion to n such that $k \to \infty$ and $(k/n) \to 0$ as $n \to \infty$ [4].

Another factor in the design of a nonparametric regression algorithm is the ability to tolerate erroneous data points in the set of observations. (This type of estimator is said to be a robust estimator.) These points, called outliers, do not conform to the model described in equation 1. The technique used to make the estimator robust is called *L-Smoothing* [8], where a fixed percentage of the observations with the largest and smallest values of t_i are eliminated from the local average.

A final issue encountered when using nonparametric regression techniques is the behavior of the estimates near the boundaries of the set of observations (i.e., no observations lie beyond the boundary). As the parameter value X approaches a boundary, the local average becomes biased, since more observation points will be on one side of point X than the other. The one-dimensional case is illustrated in Figure 4, where the estimated function $\hat{m}(x)$ will become biased near the boundary [6, 8]. To ensure accurate estimates near the boundary, a nonparametric regression technique needs to be able to compensate for this effect. A formal definition of the *k-NN* algorithm, including solutions to these issues, is presented in Appendix A.

3.2 Parameterizing Machine Performance

If the *k-NN* algorithm is used as presented above, a separate set of observations must be maintained for each machine. This condition is caused by the lack of a mechanism to translate performance differences of the machines into numeric parameters which can be incorporated into the execution time model presented in equation 1. There are two compelling reasons why it is desirable to eliminate this restriction and to form a unified set of observations. First, separate sets makes the process of adding new machines (or applications)

to the network difficult, since a few initial observations are required in each set for the algorithm to function effectively. Thus, widespread benchmarking is required to obtain such an initial set for each machine. Second, a starvation problem can exist, where a machine with few observations will tend to produce poor execution time estimates. If these poor estimates are larger than the actual execution time, it is unlikely that the scheduling algorithm will choose to execute the task on that machine. Thus, the machine will not get any new observations from which estimates could be improved.

To jointly utilize observed execution times across all machines, a method is needed to characterize the available machines using numeric parameters which can then be included in the parameter vector. This needs to be accomplished such that the distance between any two machines, in terms of their parameterization, is a rough indication of the similarity of the performance of those machines. This process can be accomplished through the use of analytic benchmarking [7].

Analytic benchmarking characterizes the performance of a machine using a series of benchmarks. In theory, each of these benchmarks should correspond to a primitive code type; code types form a basis which can exactly characterize the performance of a machine for any task. Because the construction of an ideal set of benchmarks is difficult (if not impossible), a rigorous definition of primitive code types is avoided, and instead it is assumed that a reasonable set of r benchmarks is available to approximate the performance differences between machines. These benchmarks can be used to span $\mathbb{R}_r$ where each axis corresponds to the results of one of the benchmarks, either in terms of the time required to execute the benchmark or the rate at which the machine performs iterations of the benchmark. This space will be called the *machine space*. Thus, a machine i can be represented by a point $B_i = [b_i^1 b_i^2 \cdots b_i^r]$ in the machine space, where b_i^j is the result for benchmark j on machine i. B_i will be called the *benchmark vector* for machine i.

The points in this space can then be used to construct an augmented parameter vector, which can be used with the method presented in Section 3.1. Given the r-dimensional machine space defined above, and a task with an execution time that is a function of a vector of parameters $X = [x^1 x^2 \cdots x^p]$, an augmented parameter vector in the unified parameter space can be constructed by concatenating the benchmark vector B_i and the parameter vector X, creating an $(r+p)$-dimensional parameter vector. Thus, the execution time observation of the task on machine i is associated with the augmented parameter vector $Y = [b_i^1 b_i^2 \cdots b_i^r x^1 x^2 \cdots x^p]$.

While it is desirable to use a large number of benchmarks to accurately characterize machine performance, the dimensionality of the resulting parameter vector is large. An increase in the dimensionality of the parameter vector increases the computational cost of the *k-NN* smoothing algorithm, and decreases the rate at which $\hat{m}(X)$ converges towards the true curve $m(X)$ (due to the requirement of maintaining the bias-variance tradeoff). Therefore, it is desirable to minimize the dimensionality of the benchmark vector B_i, while maximizing the amount of information it contains. Since the distance relationship between the points contains the information on the relative performance differences between the machines, this goal can be accomplished by reducing the number of dimensions in the machine space, while attempting to preserve the distance relationship between the points. Potter and Chiang [17] present an algorithm that can be used to embed the benchmark vectors B_i in an s-dimensional subspace, where $s < r$. The details of this algorithm are given in Appendix B.

This embedding creates a new machine parameter space of smaller dimension, called the *reduced machine space*. This space is combined with the parameters characterizing the input data to yield a unified parameter space, as outlined above.

3.3 Summary of the Complete Algorithm

In this section, the *k-NN* regression technique of Section 3.1 is applied to the augmented parameter vector method described in Section 3.2, to create the completed execution time estimation algorithm. The algorithm begins with a set of n previous observations of the execution time $\{(t_i, Y_i)\}_{i=1}^n$, where $Y_i = [b_{j_i}^1 b_{j_i}^2 \cdots b_{j_i}^s x_i^1 x_i^2 \cdots x_i^p]$ and j_i is the machine from which observation Y_i was obtained. Given this set, a parameter vector $X = [x^1 x^2 \cdots x^p]$ describing the input data set, and the reduced machine space $\mathbb{R}_s$ containing a point $B_j = [b_j^1 b_j^2 \cdots b_j^s]$ for each machine j, pseudocode for this algorithm can be constructed as follows.

Execution Time Estimation Algorithm:
begin
  For each candidate machine j with
    benchmark vector $B_j = [b_j^1 b_j^2 \cdots b_j^s]$
  begin
    Compute $\hat{m}(Y_j)$ and $\hat{\sigma}^2$, where
      $Y_j = [b_j^1 b_j^2 \cdots b_j^s x^1 x^2 \cdots x^p]$.
  end
  Give estimates computed above to
    matching and scheduling algorithm.
    The algorithm will return a
    machine j chosen to execute the task.
  Execute task on machine j, and
    measure the execution time t_{n+1}.
  Add observation (t_{n+1}, Y_{n+1}) to the

set of previous observations, where
$Y_{n+1} = [b_j^1 b_j^2 \cdots b_j^s x^1 x^2 \cdots x^p]$.
$n = n + 1$.
end

It can be seen that every time a given task is run on a machine in the system, a new observation is added to the set of previous observations. Thus, the quality of the predictions improves with time.

One issue which has not been addressed is the source of an initial set of observations. Since the execution time estimate is a function of the set of previous observations, at least one initial observation is required when a new task/application is introduced into the system. Thus, the task must be executed on a few selected machines in order to obtain a few initial estimates. These values are easily obtained during the development, testing, and debugging of the application.

4 Evaluation

To evaluate the performance of the methods presented in this paper, two sets of experiments were performed using real data. A machine space was constructed using the 10 benchmarks from the Byte benchmark suite [1]. These benchmarks consist of a variety of integer and floating point benchmarks. These benchmarks were executed on 16 different machines running different flavors of UNIX. The results from these benchmarks were normalized, giving all benchmarks equal weight. This 10-dimensional space was then reduced to a 3-dimensional space using the algorithm outlined in Section 3.2. The normalized result of this embedding is pictured in Figure 5. In this 16 machine environment, experiments were performed using real data obtained from the Cholesky decomposition and trial division algorithms presented in Section 2.

The first set of experiments emulates the situation where a new machine is added to the network. This experiment compares the performance of the execution time estimation algorithm when observations can and cannot be shared between machines. In this experiment, the execution time of the Cholesky decomposition algorithm was estimated on a single machine for 5 different matrix sizes. The number of observations on the machine was varied between 1 and 50. The average prediction error is compared for three different simulations. The results of these three simulations are given in Figure 6, which show how the average error in $\hat{m}(X)$ changes as the number of observations increases.

The first simulation shows the performance of the method when observations are not shared between machines. The execution time was computed to be a function of a scalar parameter: the size of the matrix. The second and third simulations show the performance of the algorithm when it is able to use observations on other machines, taking advantage of an additional 350 observations uniformly distributed across the remaining 15 machines. In the second simulation, the 3-dimensional reduced machine space is used. Thus, by using the size of the input data set and the 3-dimensional embedding as parameters, the execution time was a function of a 4-dimensional parameter vector. The third simulation used the full 10-dimensional machine space in the multidimensional algorithm. In this case, the execution time was a function of an 11-dimensional parameter vector. In all of these simulations, the value of k used to make an estimate of the execution time on a given machine j was defined to be $\lceil (0.1)(n_j)^{4/5} \rceil$, where n_j is the number of observations for machine j, which satisfies the bias-variance tradeoff requirements outlined in Section 3.1.

As shown in Figure 6, the ability to share observations between machines gives the algorithms used in the second and third simulations a significant performance advantage over the first algorithm when there are few observations from which to compute an estimate. The latter simulations produce prediction errors around 50%, versus errors around 500% using the first method. The performance difference between the two observations-sharing methods is negligible. For larger numbers of observations, all three methods perform equally, with prediction errors around 15% using a few 10's of observations. To compare the computational costs of these three algorithms, Figure 7 shows the measured CPU time of each algorithm as a function of n (the size of the entire observations set). This figure shows that using a reduced parameter space is considerably more efficient than using the full parameter space, making the reduced parameter space approach the best choice when considering both accuracy and efficiency. The algorithm was implemented using MATLAB's scripting language, and the CPU time was measured on an HP B180L workstation. While the measured CPU times are small, it is likely that a more efficient implementation could result from a conventional programming language.

In the second set of experiments, estimates were computed using the trial division algorithm on a single machine (no observation sharing was done in this experiment). As shown in Figure 2, the execution time of this algorithm is very loosely correlated to the parameter vector X. Thus, in this extreme example, the error in the execution time estimates will always be large, regardless of the number of past observations. However, these experiments demonstrate the utility of estimating the sample variance of the execution time, and using this value to bound the execution time. Two different simulations were performed, where $\hat{m}(X)$ and $\hat{\sigma}^2$ were

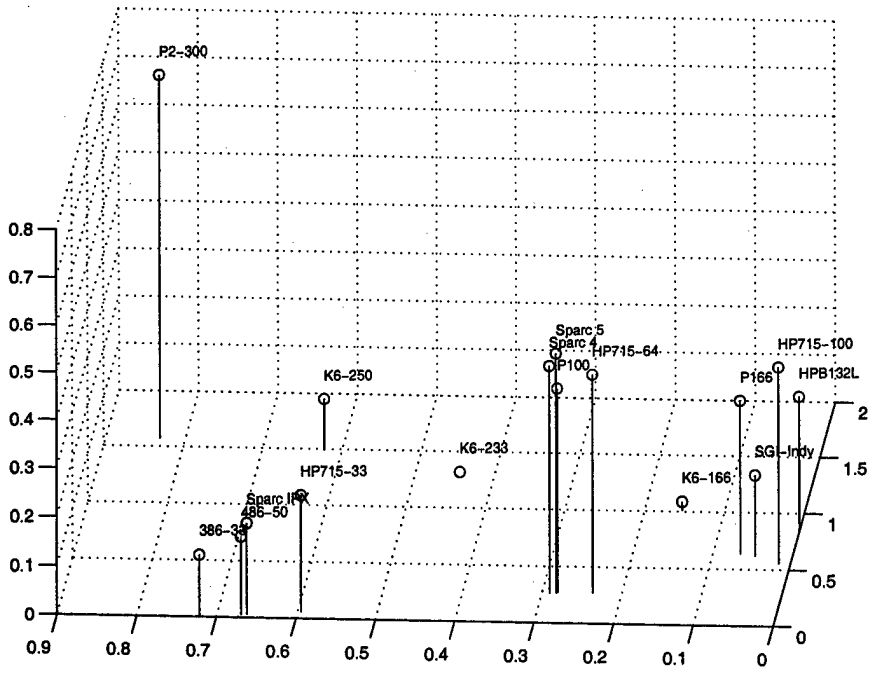


Figure 5. 3-Dimensional Distance Embedding of Machine Space.

computed for 50 evenly-spaced problem sizes using 20 and 200 observations, respectively. These results are presented in Figures 8 and 9, which show $\hat{m}(X)$ and $\hat{m}(X)+3\hat{\sigma}$. It can be seen in the figures that $\hat{m}(X)+3\hat{\sigma}$ does act as a reasonable upper bound on the execution time. Thus, the combination of both the estimated execution time and the uncertainty, σ, could be used by an appropriate matching and scheduling algorithm to make good scheduling decisions, despite the fact that the parameter vector may not contain sufficient information to obtain an accurate execution time estimate. Furthermore, Figure 8 illustrates how the execution time estimate conforms to the set of past observations, where the estimated curves form a distinct "hump" around the two observations with large execution times.

5 Conclusions

This paper presents a statistical execution time estimation algorithm for use in a heterogeneous distributed computing environment. This algorithm treats the execution time as a random variable, and makes predictions using past observations of the execution time. These estimates compensate for the properties of the input data set and the machine type, without requiring any direct knowledge of the internal operation of the task or machine. The random model allows the algorithm to determine the probable execution time of the task, even in situations where the estimate has a large amount of uncertainty. This algorithm is unique in that it is able to use observations from dissimilar machines when making predictions, through the process of analytic benchmarking. This ability greatly simplifies the process of adding new machines to the system. Furthermore, an algorithm is presented which can be used to reduce the number of parameters introduced by the analytic benchmarking process. As shown in Figure 6, experimental results indicate that this method can make accurate execution time estimates over a wide range of parameter values using a few dozen past observations.

A Appendix: k-NN Regression

The *k-NN* regression algorithm, and other statistical techniques shown in this section, are derived from the methods surveyed in the books by Härdle [8] and Eubank [6], unless noted otherwise. Given a parameter vector $X = [x^1 x^2 \cdots x^p]$ and a set of n previous observations $\{(t_i, X_i)\}_{i=1}^n$, the *k-NN* method can be formally defined as follows. Let

$$J_X = \{i : X_i \text{ is one of the } k \text{ nearest neighbors of } X\}. \quad (4)$$

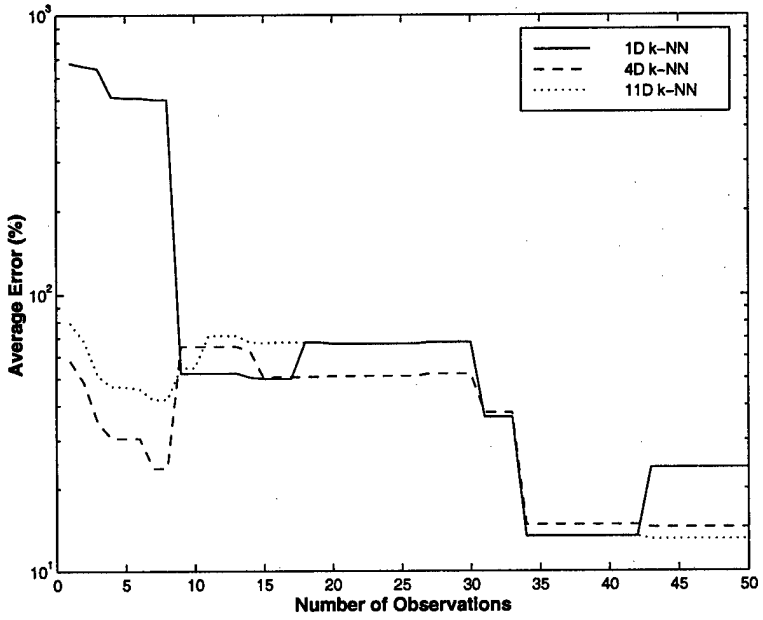


Figure 6. Average Prediction Error vs. Number of Observations.

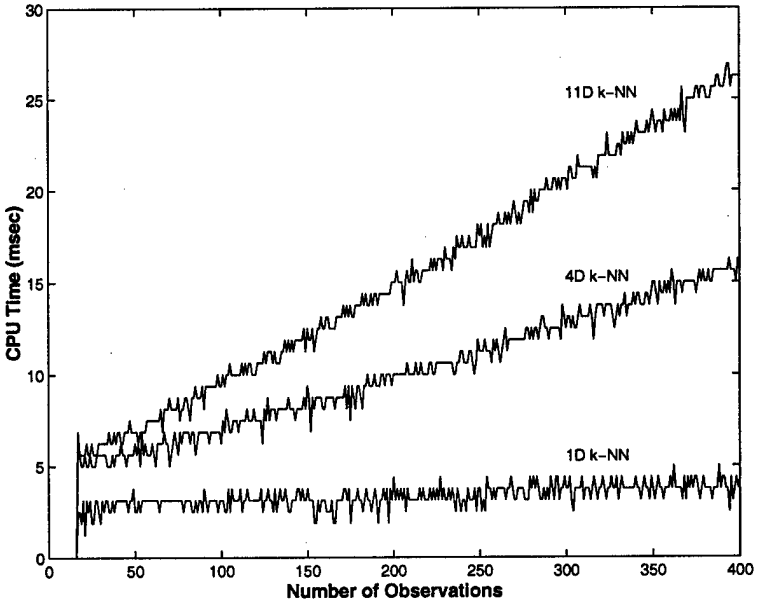


Figure 7. Computation Costs of Prediction Algorithm.

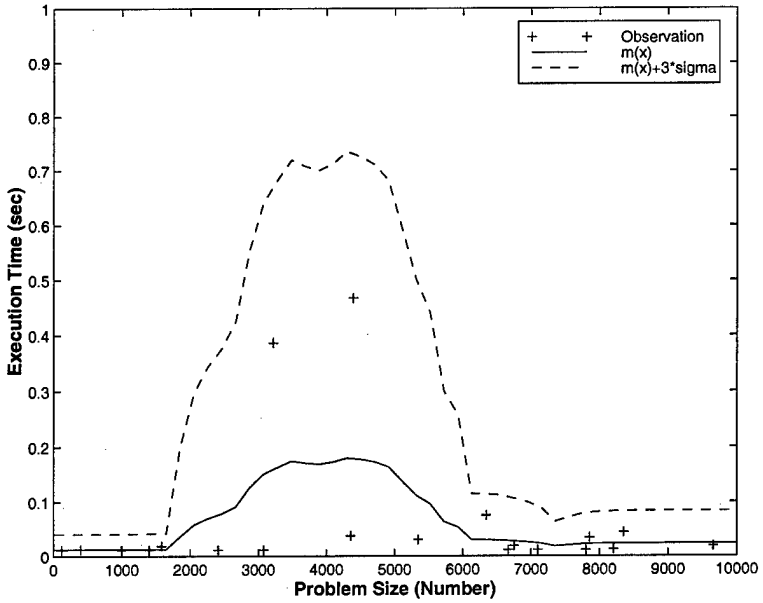


Figure 8. $\hat{m}(X)$ **and** $\hat{\sigma}^2$ **with 20 observations.**

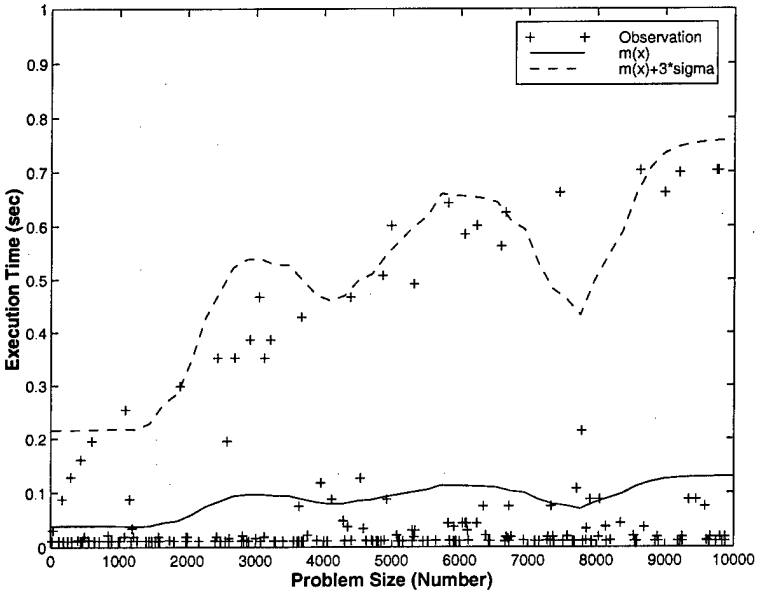


Figure 9. $\hat{m}(X)$ **and** $\hat{\sigma}^2$ **with 200 observations.**

The *k-NN* method uses the elements in J_X to form a weighted average of observations, similar to equation 2.

L-Smoothing [8] is used to make the weighted average robust (i.e., able to tolerate outliers in the data set). A fixed percentage of the observations with the largest and smallest t values are not included in the local average. *L-Smoothing* can be implemented by sorting the observations $\{(X_i, t_i)\} \in J_X$ by t_i, then computing $\hat{m}(X)$ to be

$$\hat{m}(X) = \frac{1}{k - 2\lfloor \gamma k \rfloor} \sum_{i=\lfloor \gamma k \rfloor}^{k-\lfloor \gamma k \rfloor} W_i(X) t_i \qquad (5)$$

and $\hat{\sigma}^2$ to be

$$\hat{\sigma}^2 = \frac{1}{k - 2\lfloor \gamma k \rfloor} \sum_{i=\lfloor \gamma k \rfloor}^{k-\lfloor \gamma k \rfloor} W_i(X)(t_i - \hat{m}(X))^2. \qquad (6)$$

The value of γ, where $0 \leq \gamma < 0.5$, controls the percentage of observations excluded from the average.

Next, the weight $W_i(X)$ assigned to each observation will be defined. First, consider a weighting function (also called a kernel function) for a single dimension j. It is desirable for this weighting function to give higher weights to the observations closer to the parameter value x^j. A weighting function which satisfies this condition is the Epanechnikov Kernel $K(u)$ [5, 6], where

$$K(u) = \frac{3}{4}(1 - u^2) \qquad (7)$$

and $|u| < 1$. This function is symmetric, with a maximum at $u = 0$, and can be shifted, scaled and normalized such that the point $u = 0$ corresponds to the parameter value x^j. In this way, the function gives higher weights to observations near the parameter value, and lower weights to more distant observations. To formally present this concept, the scaled Epanechnikov kernel, for dimension j, is defined to be

$$K_R^j(u) = \frac{1}{R^j} K^j\left(\frac{u}{R^j}\right). \qquad (8)$$

The Epanechnikov kernel is scaled by the factor R^j, which, for the k points in J_X, is defined to be

$$R^j = \max_{J_X}(x^j - x_i^j). \qquad (9)$$

Given these definitions, a scaled kernel function $K_R^j(u)$ can be computed for each dimension j, where $1 \leq j \leq p$. Each of these functions can then be combined into a single multidimensional kernel function $K_R(U)$, where $U = [u^1 u^2 \cdots u^p]$ and

$$K_R(U) = \prod_{j=1}^{p} K_R^j(u_j). \qquad (10)$$

Finally, the scaled multidimensional kernel can be shifted and normalized to form our weighting sequence $W_i(X)$, where, for $i \in J_X$

$$W_i(X) = \frac{K_R(X - X_i)}{\hat{f}_R(X)}, \qquad (11)$$

where $\hat{f}_R(X)$ is a normalizing factor, defined to be

$$\hat{f}_R(X) = \frac{1}{k - 2\lfloor \gamma k \rfloor} \sum_{i=\lfloor \gamma k \rfloor}^{k-\lfloor \gamma k \rfloor} K_R(X - X_i). \qquad (12)$$

This factor ensures that the sum of the weights will be one.

A modified kernel function is used for parameter values near the boundary, in order to compensate for boundary effects. This method, first presented by Rice [19], parameterizes the kernel function, which eliminates the bias near the boundaries, and yields variance near the boundaries that is the same order of magnitude as for points in the interior [19].

To define this method, first assume observations in dimension j are bounded to the interval $[a^j, b^j]$. Now, after computing the set J_X, if an observation $X_i = [x_i^1 x_i^2 \cdots x_i^p]$ with $x_i^j \leq a^j$ or $x_i^j \geq b^j$ is in the set J_X, a boundary kernel will need to be used for dimension j in equation 10. Otherwise, the regular kernel function is used. To define the boundary kernel, first define the parameter ρ^j

$$\rho^j = \begin{cases} \frac{x^j - a^j}{R^j} & \text{if } \exists X_i \in J_X : x_i^j \leq a^j \\ \frac{b^j - x^j}{R^j} & \text{if } \exists X_i \in J_X : x_i^j \geq b^j \end{cases} \qquad (13)$$

where R^j is as defined in equation 9, and x_j is the jth element of the parameter vector X. Next, define

$$\alpha^j = 2 - \rho^j. \qquad (14)$$

For the Epanechnikov Kernel, let

$$Y(\rho) = \frac{3}{4}\frac{(\rho - 1)^2}{(\rho - 2)}. \qquad (15)$$

Then, let

$$\beta^j = \frac{Y(\rho^j)}{Y(\rho^j) - \alpha^j Y(\rho^j/\alpha^j)}. \qquad (16)$$

Now a new kernel function can be defined to be

$$K_\rho^j(u) = (1 - \beta^j)K^j(u) + (\beta^j/\alpha^j)K^j(u/\alpha^j). \qquad (17)$$

The function K_ρ^j can be directly substituted for the function K^j defined in equation 8.

B Appendix: Embedding Points in an s-dimensional Space

Potter and Chiang [17] present an algorithm to embed points in a lower dimensional space in a manner which attempts to preserve the distance relationship between the points. This method begins with the r-dimensional machine space defined above, which contains m points $B_1, B_2, \ldots, B_m \in \mathbb{R}_r$ representing the available machines. As mentioned above, the distance relationship between these m points will be represented using a Euclidean distance matrix $D_r = \{d_{ij}\} \in \mathbb{R}_{m \times m}$ (a real $m \times m$ matrix) in $\mathbb{R}_r$, where

$$d_{ij} = -\frac{1}{2} \| B_i - B_j \| \tag{18}$$

and $\| Z \| = (\sum_{l=1}^{r} z_l^2)^{1/2}$.

The goal of this algorithm is to find a set of m points in $\mathbb{R}_s$, where $s < r$, with distance matrix D_s providing the best-fit to D_r. The algorithm operates as follows.

1. Compute the Euclidean distance matrix D_r from the m points in $\mathbb{R}_r$.

2. Compute the orthogonal projection matrix P, which is defined to be

$$P = I - \frac{1}{m} e e^T, \tag{19}$$

   where $e = [11 \cdots 1]$ is a vector in $\mathbb{R}_m$, and I is an $m \times m$ identity matrix.

3. Construct the matrix

$$A = PDP^T. \tag{20}$$

4. Diagonalize A with an orthogonal matrix U and a diagonal matrix Λ, such that

$$A = U \Lambda U^T. \tag{21}$$

5. Form $\hat{\Lambda}$ by retaining the s largest eigenvalues in Λ and setting the rest to zero.

6. Compute the matrix

$$C = U \hat{\Lambda}^{1/2}. \tag{22}$$

The rows of the matrix C will give the coordinates of the m points in $\mathbb{R}_s$ which have a distance relationship closest to that of the original points [17].

References

[1] BYTEmark benchmark documentation. *BYTE Magazine Web Page (http://www.byte.com/ bmark/bdoc.htm)*, 1998.

[2] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proc. of the 1998 Heterogeneous Computing Workshop*, pages 79–87, Orlando, FL, Mar. 1998. IEEE Computer Society Press.

[3] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. *IEEE Trans. Software Engineering*, 15(12):1579–1586, Dec. 1989.

[4] L. P. Devroye. The uniform convergence of nearest neighbor regression function estimators and their application in optimization. *IEEE Trans. Information Theory*, IT-24(2):142–151, Mar. 1978.

[5] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability and Its Applications*, 14:153–158, 1969.

[6] R. L. Eubank. *Spline smoothing and nonparametric regression.* M. Dekker, 1988.

[7] R. Freund. Optimal selection theory for superconcurrency. In *Proceedings of the 1989 Supercomputing Conference*, pages 13–17. IEEE Computer Society Press, 1989.

[8] W. Härdle. *Applied nonparametric regression.* Cambridge University Press, 1990.

[9] C.-J. Hou and K. G. Shin. Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems. *IEEE Trans. Computers*, 43(9):1076–90, Sept. 1994.

[10] M. A. Iverson, F. Özgüner, and G. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *Proc. of the 1996 High Performance Distributed Computing Conference*, pages 263–270, Syracuse, NY, Aug. 1996.

[11] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang. Heterogeneous supercomputing: Problems and issues. In *Proc. of the 1992 Workshop on Heterogeneous Processing*, pages 3–12. IEEE Computer Society Press, Mar. 1992.

[12] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. Heterogeneous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[13] T. Kidd, D. Hensgen, L. Moore, R. Freund, D. Charley, M. Halderman, and M. Janakiraman. Studies in the useful predictability of programs in a distributed and homogeneous environment. *The Smartnet Home Page (http://papaya.nosc.mil:80/SmartNet/)*, 1995.

[14] W. R. King. A stochastic personnel-assignment model. *Operational Research*, 13(1):67–81, Jan. 1965.

[15] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, and D. K. Watson. Estimating the distribution of execution times for SIMD/SPMD mixed-mode programs. In *Proc. of the 1995 Heterogeneous Computing Workshop*, pages 35–46. IEEE Computer Society Press, Apr. 1995.

[16] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. *IEEE Computer*, 24(1):18–29, Jan. 1991.

[17] L. C. Potter and D. Chiang. Distance matrices and modified cyclic projections for molecular conformations. In *Proceedings of the 1992 Inter. Conf. on Acoustics, Speech, and Signal Processing*, pages 173–176. IEEE Press, 1992.

[18] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *Proc. of the 1994 ACM Conference on LISP and functional programming*, pages 65–78. ACM Press, June 1994.

[19] J. Rice. Boundary modification for kernel regression. *Communications in Statistics—Theory and Methods*, 13(7):893–900, 1984.

[20] H. J. Siegel. Heterogeneous computing. *Annual Research Summary 5.92, http://ece.www.ecn.purdue.edu/Researchsummary/Section5/sec5_92.html*, 1994.

[21] M. Tan and H. J. Siegel. A stochastic model of a dedicated heterogeneous computing system for establishing a greedy approach to developing data relocation heuristics. In *Proc. of the 1997 Heterogeneous Computing Workshop*, pages 122–34, Geneva, Apr. 1997. IEEE Computer Society Press.

[22] J. Yang, I. Ahmad, and A. Ghafoor. Estimation of execution times on heterogeneous supercomputer architectures. In *the 1993 Inter. Conf. on Parallel Processing*, volume 1, pages 219–226. CRC Press, Aug. 1993.

[23] T. Yang and A. Gerasoulis. DSC: Scheduling tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(6):951–967, Sept. 1994.

Michael Iverson received the B.S. degree in Computer Engineering at Michigan State University in 1992, and the M.S. degree in Electrical Engineering at The Ohio State University in 1994. He is currently researching topics in heterogeneous distributed computing for his Ph.D. dissertation. Mr. Iverson has also developed Internet video conferencing systems and wireless networking systems for Ohio State. Upon completion of his degree, he will be employed at Iverson Industries Inc. of Wyandotte, Michigan.

Füsun Özgüner received the M.S. degree in electrical engineering from the Istanbul Technical University in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975. She worked at the I.B.M. T.J. Watson Research Center with the Design Automation group for one year and joined the faculty at the Department of Electrical Engineering, Istanbul Technical University in 1976. Since January 1981 she has been with The Ohio State University, where she is presently a Professor of Electrical Engineering. Her current research interests are parallel and fault-tolerant architectures, heterogeneous computing, reconfiguration and communication in parallel architectures, real-time parallel computing and parallel algorithm design. She has served as an associate editor of the IEEE Transactions on Computers and on several conference program committees.

Lee C. Potter received the B.E. degree from Vanderbilt University and M.S. and Ph.D. degrees from the University of Illinois, Urbana, all in electrical engineering. Since 1991 he has been with the Department of Electrical Engineering at The Ohio State University where he is currently Associate Professor. His research interests include statistical signal processing, inverse problems, detection, and estimation, with applications in radar target identification and ultra wide-band systems. Dr. Potter is a 1993 recipient of the Ohio State College of Engineering MacQuigg Award for Outstanding Teaching.

Simulation of Task Graph Systems in Heterogeneous Computing Environments

Noe Lopez-Benitez and Ja-Young Hyon
Department of Computer Science
College of Engineering
Texas Tech University
Lubbock, Texas 79409-3104
nlb@ttu.edu

Abstract

This paper describes a simulation tool for the analysis of complex jobs described in the form of task graphs. The simulation procedure relies on the PN-based topological representation of the task graph that takes advantage of directly modeling precedence constraints and other characteristics inherent in Generalized Stochastic Petri Nets (GSPN). The GSPN representation is enhanced with enabling functions that govern the sequence of firings of transitions representing execution of tasks. The regulated flow of activity is carried out observing not only precedence constraints but specific allocation heuristics and communication delays. The tool is useful in evaluating different heuristics described by the corresponding implemented algorithm, or using a deterministic timespan given by a Gantt chart.

1. Introduction

Task graphs represent general computation jobs which have been decomposed into modules called tasks that are executed according to some precedence constraints. Task graphs are a well known tool to study performance issues of complex jobs. A direct solution technique for series-parallel task graphs is reported in [1]; an average completion time of the overall job is derived assuming no restrictions exist on the number and architecture of processing units and with no regard to allocation schemes. Execution times of fork-join parallel programs in multiprocessor environments is discussed in [2]. An approach based on multiplication/convolution is applied to Heterogeneous Computing Systems (HCS) at coarse and fine levels of granularity in [3]. Also, in [4] performance prediction of fork-join task graphs is addressed, where the residence times of each task are estimated in terms of service demands and queuing delays; based on these estimations, the task graph is then systematically reduced.

Markov-based solutions of task graph systems have been reported in [5] and [6]; although limited to relatively small task graphs, a Markov-based solution is used for the analysis of scheduling policies in [6]. Since Stochastic Petri Nets (SPN) provide a natural representation of parallelism and synchronization their use spawns applications from individual parallel and concurrent programs to distributed applications and multi-processor systems [7, 8, 9]. SPN's can be used to directly capture the topological information of a task graph and provide a systematic way for applying factors such as processor heterogeneity, allocation schemes, communication costs, and random execution times. Also, a SPN-based solution can be applied to arbitrary graphs which are acyclic but not necessarily series-parallel [10]. SPN-based tools automatically generate Markov models that represent the execution process of complex task graphs where each state is given by the number of tasks executing in parallel. These models are then solved to compute system performance characteristics such as a distribution of the overall completion time.

When the job represented by a task graph is executed on the processing elements of a HCS, estimating the overall completion time becomes an optimization problem involving the mapping of tasks to processors such that completion time is minimized. Mapping tasks to processing units is a hard problem and several heuristics have been proposed in the literature. However, before choosing the most effective heuristic a method must be available for computing an expected completion time and deriving execution distributions for any given task graph, HCS, and allocation heuristic. The methodology reported in [10] to solve complex task graphs using SPN's is not in itself an optimization technique, but it can be used in conjunction with optimization techniques which attempt to search a space of completion time distributions. However, Markov-based numerical solutions are limited to exponential distributions and often involve a large state space.

0-7695-0107-9/99 $10.00 © 1999 IEEE

Consequently, the solution process may be unstable and subject to stiffness problems rendering inaccurate results. Discrete event simulation can use the framework provided by SPN's [11] and circumvent the limitations encountered in the solution of Markov-based models. The work reported in this paper uses the SPN-based topological representation of task graph systems just as in [10] but applies discrete-event simulation to obtain execution time distributions and estimates of the Mean Time to Completion (MTTC) of the jobs represented. Thus a common model based on SPN's is used to drive a discrete event simulation of the overall job. The method can be used to analyze and compare several assignment heuristics given either the algorithm or a Gantt chart of specific assignment cases. To illustrate the use of the tool several assignment heuristics are evaluated and compared.

The next section of the paper introduces the notation and parameters used. Basic concepts on Petri nets are introduced in section three and their application to describe task graphs is given in section four. Section five deals with the simulation methodology. A brief discussion on allocation heuristics is given in section six. The insertion of communication delays is discussed in section six. Simulation algorithms are presented in section seven. Lastly, applications of the tool are discussed.

2. Parameters and Notation

Throughout the paper the following notation is used to describe the simulation tool and related issues.

- a task graph $G(T, E)$ where the vertex set $T = \{T_1, T_2, \ldots, T_k\}$ consists of k tasks which compose some overall job and the edge set E consists of ordered pairs from T which correspond to data or control dependencies. The topology of T is described in detail by the following:
  - an in-degree vector $D = [d_1, d_2, \ldots, d_k]$ where d_i is the number of tasks which must complete before T_i may initiate execution.
  - an out-degree vector $H = [h_1, h_2, \ldots, h_k]$ where h_i is the number of tasks which are spawned after the completion of T_i.
  - a task graph structure $TG[i][j]$, $1 \le i \le k$, $1 \le j \le h_i$ where $TG[i]$ is an array specifying the h_i tasks which are spawned by the completion of T_i; thus, the ordered pair $(T_i, TG[i, j]) \varepsilon E$.
- a $k \times k$ matrix $pkt[i, j]$, $1 \le i, j \le k$ where $pkt[i, j]$ is the average number of data packets of standard size that is sent from T_i to T_j. Alternatively, these can be specified as edge weights for the elements of E.
- a priority vector $W = [w_1, w_2, \ldots, w_k]$ which induces a sequential ordering of any ready tasks assigned to the same processor; these priorities may be taken from the indices of the tasks, e.g. $w_i = k - i$, or they may be randomly or determined according to the assignment heuristic employed.
- a set $P = \{P_1, P_2, \ldots, P_n\}$ consisting of n processors composing a heterogeneous suite.
- a $k \times n$ execution time matrix $B[i, j]$, $1 \le i \le k$, $1 \le j \le n$ where b_{ij} is the average execution time of T_i on P_j.
- an $n \times n$ communication time matrix $C[r, s]$, $1 \le r, s \le n$ where each entry c_{rs} is the average communication time to transfer a data packet of standard size from P_r to P_s.
- a $k \times n$ static allocation matrix $A[i, j]$, $1 \le i \le k$, $1 \le j \le n$ where entry $a_{ij} = 1$ if T_i has been allocated to P_j, and 0 otherwise.

3. Basic Petri Net Concepts

A Petri net (PN) is a directed, weighted, and bipartite graph [12]. PN's are bipartite in that nodes are of two types, *places* and *transitions*, with arcs occurring either from places to transitions or from transitions to places. When an arc is from a place p to a transition t, then p is an input place of t; a place p is an *output place* of t if an arc proceeds from t to p. Places and transitions are represented pictorially by circles and thin rectangles, respectively. A third component of any PN are tokens which reside in places; pictorially, tokens are represented by dots within the perimeters of places. Tokens are transferred from one place to another by the firing of transitions. When a transition t fires, tokens are removed from all input places of t and placed in the output places of t; thus, enforcing a logical flow of activity throughout the net. A transition can fire if it is enabled, i.e., if all of its input places possess at least one token. An arc may be weighted where the weight specifies the number of tokens which must reside in an input place in order for a transition to be enabled, or the number of tokens placed in an output place by the firing of an enabled transition; if the weight is unspecified then it is assumed to be one. PN's and their dynamic behavior can be captured in mathematical notation via state vectors. Given a PN with k places, a marking q of the PN is denoted by M_q; a marking is described by a k – vector whose ith component denotes the number of tokens in place p_i; an initial marking of the PN is denoted by M_0. A particular PN with an underlying graph N is denoted (N, M_0). The reachability graph of a PN is a graph $G_R(M, \Delta)$ where the vertex set M is the set of all possible markings for the PN and the edge set Δ consists of all possible transition firings transforming one marking into another.

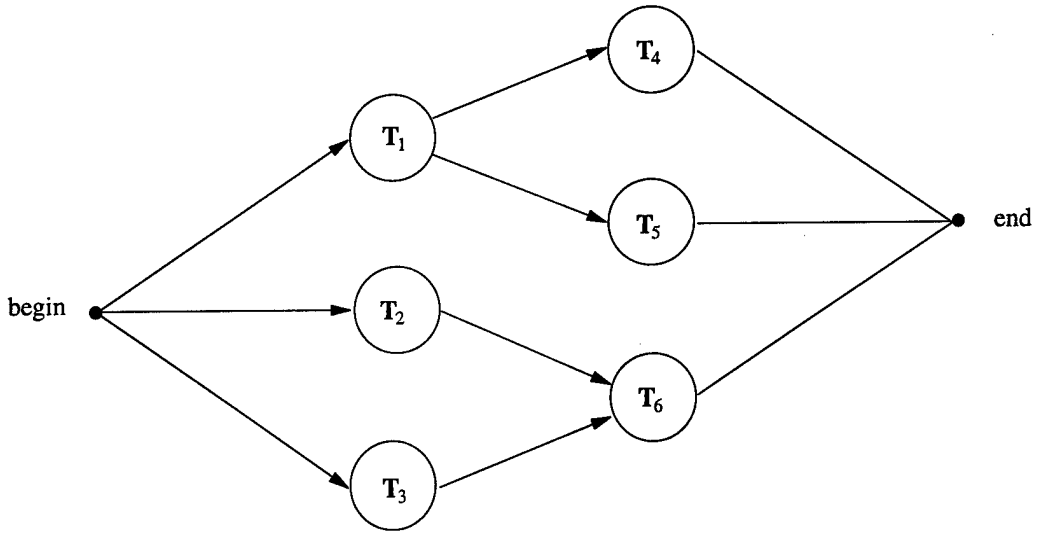


Figure 1. A simple task graph

Stochastic Petri nets are PN's in which there is an exponentially distributed delay time between the enabling and firing of transitions. The reachability graph of a bounded SPN is isomorphic to a finite Markov chain (MC) [13]; in particular, the markings of the reachability graph comprise the state space of a MC and the transition rate between any two states X_i and X_j is the sum of all firing delays for transitions transforming M_i into M_j. *Generalized stochastic Petri nets* (GSPN) have been proposed [14] in which transitions are of two types: *timed* transitions which have the exponentially determined firing rates and *immediate* transitions which have no firing delay and have priority over any timed transition. *Enabling functions* are marking-dependent functions which can be defined on each transition as a switching mechanism. Transition priorities (timed vs. immediate) and enabling functions are logically equivalent extensions of SPN which endow them with the full computational power of Turing machines [15]. In this paper the notion of GSPN is used.

4. GSPN Models of Task Graphs

Task graphs are assumed to be series-parallel for several approaches to performance evaluation [16] and optimization [17]; however, this limitation is avoided in the PN-based methodology of this work. Fig. 1 shows a simple task graph which will be used to illustrate the translation of task graphs into GSPNs. The translation of a task graph into a GSPN begins with the association of each task T_i with a place/timed transition pair, p_i and t_i. Fig. 2 shows the GSPN corresponding to the task graph in Fig. 1. Auxiliary places xp_0 and xp_1 and immediate transitions it_0 and it_1 are used to enforce initiation and completion conditions, respectively, for the overall job. The presence of at least one token in a place may represent the fulfillment of all preconditions for the initiation of the task. The firing of a timed transition represents the completion of execution of the corresponding task. The delay time of each transition corresponds to the exponentially distributed execution time of the task. A place p_i can be associated with the in-degree d_i to enforce precedence constraints. Initially, the presence of a token in xp_0 enables it_0; the firing of it_0 represents the initiation of an execution cycle. The presence of three tokens in xp_1 and the firing of it_1 indicates that an execution cycle has been completed. Timed transitions in the GSPN model in Fig. 2 will fire once enabled. Beginning with an initial marking M_0 a sequence of markings can be generated to form a reachability graph. The set of markings generated correspond to the possible execution states of the system, where a system state is defined by the tasks which are executing concurrently. If firing times are exponentially distributed the set of markings generated corresponds to a Markov chain that can be solved using well known tools such as SPNP [18] or SHARPE [1].

Consider some marking M_i in which task T_6 should be ready to run. To make this possible, both T_2 and T_3 must have finished execution; this will be indicated by the presence of two tokens in p_6, i.e. $x_i(p_6) = 2$. To capture this precedence constraint it suffices to associate each input arc into a timed transition with a weight corresponding to the in-degree of each node in the task graph. Alternatively, the in-degree vector is associated with marking-dependent enabling functions.

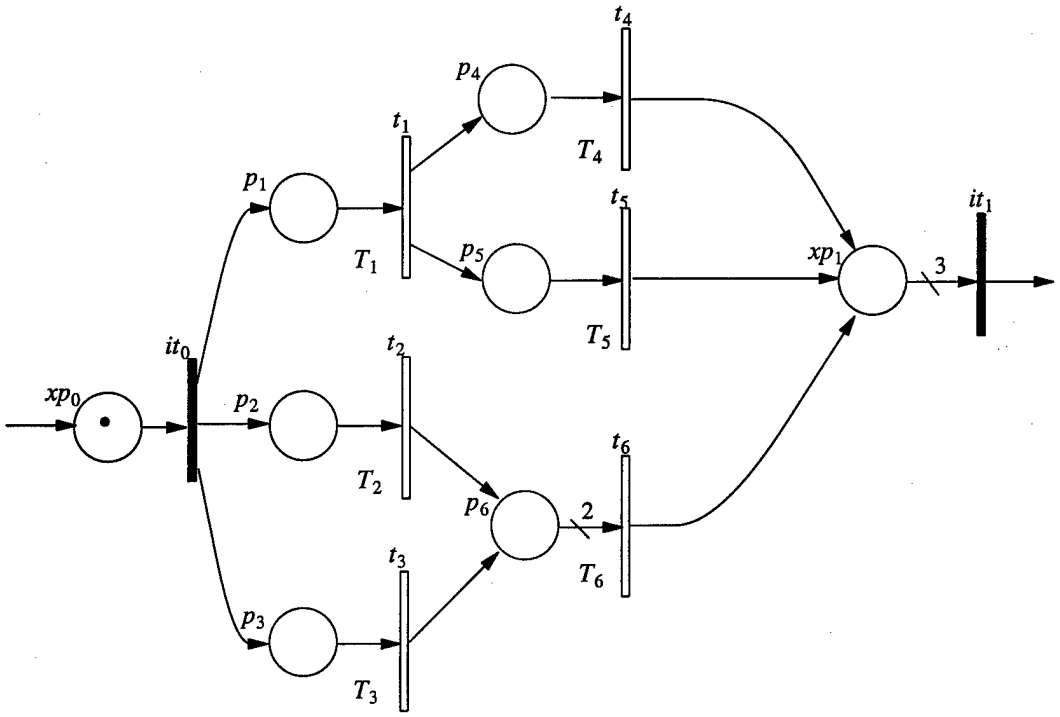


Figure 2. GSPN model of the task graph from Fig. 1

5. Simulation Methodology

In a SPN model, the firing of transitions represents the occurrence of events, in this case, execution of tasks. To simulate execution of tasks [11], a clock is set for each newly enabled transition to keep track of the execution time until the transition fires. The simulation procedure must also check for precedence constraints, availability of processors, and priority of tasks. When a transition is enabled its firing time is generated as a random variate from a selected distribution. Firing times are recorded by associating clocks to transitions. The PN-based simulation procedure takes place observing the following major steps:

1) Check for newly enabled transitions,

2) Generate firing firing times, and

3) Update clocks.

5.1. Enabling Functions

A transition t_i is enabled when conditions Q_i, V_i, and Z_i are satisfied. An entry of the enabling vector $F = [f_i], 0 \le i \le k$ is evaluated such that if:

$$f_i = Q_i V_i Z_i$$

evaluates to one and t_i can fire.

Condition Q_i checks for precedence constraints, that is, when the number of tokens m_i in place p_i is equal to the in-degree d_i of the vertex representing task T_i, its precedence constraints are met, i.e.,

$$Q_i = \begin{cases} 1 & \text{if } m_i = d_i \\ 0 & otherwise \end{cases}$$

Condition V_i checks for allocation and availability of processors. To check for allocation suffices to examine the *ith* row of matrix A for $a_{ij} = 1$ and then verify if processor j is free. Let a binary vector $FREE = [free_j], 0 \le j \le n-1$ keep track of which processors are currently free, then

$$V_i = a_{ij} free_j$$

If more than one transition satisfies condition Q and V and their corresponding tasks are allocated to the same processor, only one transition should be enabled (only one task should execute) even though these tasks could execute in parallel. The task with the highest priority is chosen using the priority vector W. Let *Rdy* be the set of transitions representing parallel tasks allocated to the same processor. That is, the set of transitions that could be enabled from a current marking M, then

$$Z_i = \begin{cases} 1 & \text{if } w_i = \max_{j \varepsilon Rdy} \{w_j\} \\ 0 & otherwise \end{cases}$$

Note that these functions could be easily implemented by incorporating additional places and transitions to the model in Fig. 2. For example the presence of a token in a dedicated place can be used to model the availability of a processor and to derive statistical measurements on the usage of that processor [19]. Also additional immediate transitions can be used to model task priorities. It can be argued that additional modeling elements may obscure the representation of a task graph and although they are useful, they become transparent to the user when dealing with large complex models. We find the addition of places to model processing elements and their interconnections useful for the case of analyzing the behavior of systems running several jobs modeled by different task graphs or several instances of the same job in an effort to capture the load of the system, resource contention and usage. In our case the effect of external load is reflected in the execution time of each subtask. The use of enabling functions keeps the model simple and the simulation code relatively simple as well.

5.2. Firing times

If a timed transition is enabled, a firing time is generated using a firing transition rate given in terms of the average execution times of each task obtained from matrix B. Random variates are generated from three possible distributions: exponential, normal, and uniform. The values given by matrix B are used according to the distribution function selected. Uniform and normal functions require a second value that must be provided by the user. If $B1$ denotes the first matrix given as the execution matrix B then $B2$ denotes a second matrix provided by the user for the case of normal and uniform distributions. For exponential and normal distributions $b1_{ij}$ provides the average execution time. For normal distributions the matrix $B2$ provides the standard deviation ρ_{ij}. In the case of a uniform distribution, matrix $B1$ provides the starting point $b1_{ij}$ and matrix $B2$ provides the ending point $b2_{ij}$. These values are used to calculate the mean as $(b1_{ij} + b2_{ij})/2$. A pseudo-random number u is generated from $U(0, 1)$. A firing time x_{ij} associated to transition t_i is generated for each distribution as follows:

i). Exponential distribution, $\exp(b1_{ij})$:

$$x_{ij} = -b1_j \times \ln(u)$$

ii). Normal distribution, $N(b1_{ij}, b2_{ij}^2)$:

$$x_{ij} = (\sum_{a=1}^{a=12} u_a - 6) \times b2_{ij} + b1_{ij}$$

iii). Uniform distribution, $U(b1_{ij}, b2_{ij})$:

$$x_{ij} = u \times (b2_{ij} - b1_{ij}) + b1_{ij}$$

5.3. Clock Update

A local clock that keeps track of firing times and a global clock is used to record the overall completion time. When a timed transition is enabled a local clock is set to the generated firing time to indicate the remaining time until the transition fires. A global clock is denoted as C and local clocks are represented by a vector $LC = [lc_i]$, $0 \le i \le k-1$ where lc_i is the local clock associated to transition t_i.

Since local clocks indicate remaining times, they are discarded when they reach 0 time units and the corresponding transitions fire. At the moment a transition fires, the global clock and local clocks are updated. The global clock update is performed by adding the minimum local clock time min_t to the global clock C; min_t is taken from the set of enabled transitions that have not yet fired. The following expressions are used to update all clocks.

$$C = C + min\_t$$

$$lc_i = lc_i - min\_t$$

where $min\_t = \min_i \{lc_i\}$, $0 \le i \le k-1$. Once the last transition fires, the global clock C indicates the overall completion time.

6. Heuristics

Different allocation heuristics can be evaluated by mapping them into the allocation matrix A. To illustrate the use of the simulation methodology discussed in this paper four static allocation heuristics are evaluated and compared.

1. *Shortest Estimated Execution Time First (SEETF).* In this scheme [20, 21, 22] task T_i is selected at random from the task set and assigned to the processor that executes T_i faster. The elements of the task allocation matrix from the SEETF algorithm are determined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } b_{ij} = \min_j \{b_{ij}\} \\ 0 & otherwise \end{cases}$$

2. *Minimum Finish Time (MFT).* In this allocation scheme [22], task T_i is also selected randomly from a topologically sorted task set, i.e. taking into account the precedence constraints between tasks. The selected processor is the one that minimizes the finish time of a task in a deterministic simulated execution, where the finish time of a selected task T_i is given by the

minimum sum of its execution time b_{ij} and the next time instance in which processor P_j becomes a free processor.

$$a_{ij} = \begin{cases} 1 & \text{if } \min_j \{b_{ij} + \textit{time until } P_j \textit{ is free}\} \\ 0 & \textit{otherwise} \end{cases}$$

Note that all tasks are selected randomly but restricted to those tasks whose predecessors have already been allocated.

3. *Largest Task First (LTF)* [22]. The selection of tasks is based on service demands. The task with the largest service demand is selected first, or alternatively the task with the largest execution time is selected first. Thus:

$$a_{ij} = \begin{cases} 1 & \text{if } b_{ij} = \max_i \{b_{ij}\} \\ 0 & \textit{otherwise} \end{cases}$$

A processor P_j is selected randomly.

4. *Most Data Task First (MDTF)* This scheme selects the task that generates most data. The data generated by a task T_i is determined in terms of the number of data packets going out, that is:

$$pkt_i = \sum_{j=1}^{k} pkt_{ij}$$

Thus, the construction of the allocation matrix proceeds as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } pkt_{ij} = \max_i \{pkt_i\} \\ 0 & \textit{otherwise} \end{cases}$$

and in this case also the processor P_j is selected randomly.

7. Communication Delays

As in [10], two approaches are presented based on two types of interconnection networks: (a) a high-performance network characterized by high-connectivity and parallel communications and (b) a bus-oriented network with low-connectivity. In both cases, output data is assumed to be accumulated in a buffer during task execution and transmitted after task completion.

7.1. Modeling High-performance Communication Networks

High-performance communication networks can be characterized as expensive systems in which inter-node communication takes place on dedicated, point-to-point links. Data intended for each successor is written to a separate buffer. Furthermore, each processor may be coupled with a front-end communication processor which enables parallel communication. In terms of a task graph, once a given task completes, successor tasks experience an initiation delay equal to the data transfer time for all intended packets; ideally, any successor task allocated to the same processor as the parent task should be able to begin execution immediately after the completion of the parent task.

The properties of such a high-performance network can be modeled in a GSPN by inserting additional place/timed-transitions to represent each individual communication; augmentation of the task graph with communication nodes has been proposed for CTMC-based analysis [23] and at the SPN level [24]. Each timed-transition inserted is associated with an exponentially distributed delay whose parameter is the average communication time between the host processors. Thus, given a completed task T_i allocated to processor P_r and a successor task T_j allocated to P_s, the average communication rate assigned to the transition modeling the transfer of data is given by:

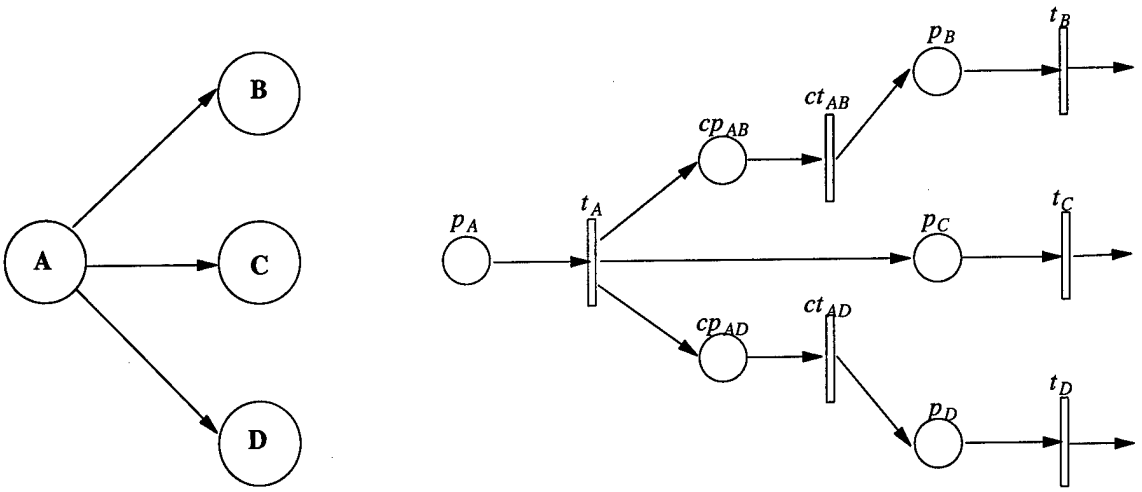
$$\delta_{ij} = \frac{1}{c_{rs}\, pkt_{ij}}$$

Fig. 5a illustrates a segment of some task graph in which Task A spawns tasks B, C, and D. Suppose the four tasks are allocated to three processors such that A and C are allocated to one processor, and B and D are allocated to the other two processors, then the resulting GSPN for Case 1 would be as shown in Fig. 5b. Note the insertion of place/transition pairs between A and B and A and D to represent the individual

In terms of simulation, communication delays are determined from a distribution function using the average delay δ_{ij} and associating a local time to communication tasks.

7.2. Modeling Bus-Oriented Networks

In interconnection networks characterized by low-connectivity, groups of processors may have to share common communication links, as is the case with a bus-oriented architecture. Also, in lower cost systems processors may be forced to expend computation cycles on communication processing. If, additionally, output data packets for successor tasks are queued up in a single buffer in some random ordering and transmitted on a FIFO basis, then it is highly unlikely that a successor task will receive all of its packets before any other successor task. In terms of the example in Fig. 5a, if the processor to which task A is allocated must broadcast packets in random order to the processors associated with tasks B, C, and D , then it is reasonable to assume that on average B, C, and D will experience uniform initiation delay.



a) Segment of a task graph

b) SPN with communication nodes

Figure 5. GSPN model assuming a high-performance network

Such behavior can be reflected in the GSPN by simply modifying the rate function governing the firing of the transitions associated with each task. In this case, no extra nodes are inserted in the PN model. Rather, the firing delay of each transition is increased by the sum of communication costs associated with each successor task. Let T_i be allocated to P_j where completion of T_i spawns $m = h_i$ tasks $T_{q_1}, T_{q_2}, \ldots, T_{q_m}$ which are allocated to processors $P_{y_1}, P_{y_2}, \ldots, P_{y_m}$. Then a modified firing rate for transition t_i is given by:

$$\tilde{\lambda_i} = \frac{1}{\mu_{ij} + \sum_{k=1}^{m} c_{jy_k} pkt_{iq_k}}$$

This new value is then used to determine execution times from the distribution function of choice with the value of μ_{ij} determined accordingly. In reality a given network may be heterogeneous with respect to interconnection capabilities. In this case the GSPN model can be systematically constructed to appropriately model each segment of the network, reflecting the different sets of assumptions mentioned above. The net result is that the simulation process uses a GSPN representation with dynamically determined transition rates and enabling functions capturing the full interplay of task precedence relationships, allocations specifications, availability of idle processors, diverse execution rates across a heterogeneous suite, and communication delays.

8. Simulation Algorithms

A simulation algorithm based on the PN-based topological description of task graphs is now described. The algorithm generates the MTTC and a tabulation to plot the cumulative probability distribution of the execution time. The following steps summarize the simulation process for the case in which no communication delay is taken into account:

1) Initialize the global clock C and the initial marking M_0.

2) Check for newly enabled transitions. In the absence of newly enabled transitions go to step 5).

3) For each enabled timed transition t_i generate firing time x_{ij}.

4) For each enabled timed transitions, set the local clock lc_i to $lc_i = x_{ij}$.

5) Find the minimum local clock *min_t*

6) Fire the transition with the minimum clock *min_t*. Once a timed transition fires, the corresponding task completes execution and the host processor is released.

7) Update global clock C and local clocks lc_i. Notice that by firing transitions with the minimum remaining time equal to *min_t*, its $lc_i = 0$ and removed from the set of lc_i's. The firing of the last timed transition ends the current cycle. A new cycle begins at step 1) by resetting the initial marking M_0 and the global clock C.

8) Update the marking record and repeat from step 2).

The above procedure is also used for the case of low-performance networks where the firing rates are modified accordingly. To take into account transfer delays in a high-performance network some modifications are needed. Let cc_{ih} denote the communication clock between task T_i and task T_h. Note that transition t_i is a transition that has already fired, that is, the corresponding task T_i is in the process of transferring data. After transfer is complete, a token travels to output place p_h. The set of communication clocks cc_{gh} is also compared with local clocks lc_i to determine the minimum time min_t. Note that the set of lc_i's corresponds to transitions t_i that have been enabled but are not yet transferring data. If the min_t selected corresponds to a local clock lc_i, then transition t_i fires, else, the min_t corresponds to a communication clock and a token is now transferred to a destination place p_h. Steps 1) to 4) are the same and the rest of the algorithm is modified as follows:

5) Find the minimum local clock: $min\_t = \min_{ih}\{lc_i, cc_{ih}\}$.

6) Update global clock C, local clocks lc_i, and communication clocks cc_{ih}:

$$C = C + min\_t$$

$$lc_i = lc_i - min\_t$$

$$cc_{ih} = cc_{ih} - min\_t$$

7) If min_t corresponds to a local clock lc_i, then:

7.1) Transition t_i fires. Tokens are removed from the input places and the corresponding processor is released.

7.2) If t_i is the last transition, then stop the cycle.

7.3) Generate communication delays and set communication clocks to $cc_{ih} = \frac{1}{\delta_{ih}}$.

8) If min_t corresponds to a communication clock then transfer a token to output place p_h.

9) Update the marking record and go back to step 2).

9. Applications

A hypothetical 13-node task graph is shown in Fig. 6. This graph was used in [10] to illustrate a PN-based numerical approach to the solution of complex task graphs. The simulation procedure is applied to the task graph and compared with the results rendered by the SPNP tool [18]. The static allocation scheme used maps tasks to processors such that a task T_i is assigned to processor P_j where $j = i \ modn$. The edge weight shown in Fig. 6 correspond to the number of standard sized packets generated and sent to successor tasks.

The following matrix B specifies the spectrum of execution times for each task across six processing units in the system in standard time units per execution:

$$B^T = \begin{bmatrix} .9 & 2 & .3 & 1 & .3 & 4 & 2 & 1 & 3 & 2 & .3 & .2 & .1 \\ .3 & 4 & .3 & 1 & .3 & 5 & 2 & 1 & 3 & 4 & .5 & .5 & .1 \\ .5 & 1 & .5 & 1 & .3 & 5 & 2 & 1 & 4 & 2 & 5 & .2 & .3 \\ .5 & 2 & .2 & 2 & .3 & 5 & 2 & 2 & 2 & 2 & .5 & .2 & .1 \\ .5 & 2 & .3 & 1 & .6 & 5 & 3 & 1 & 3 & 2 & .5 & .2 & .1 \\ .5 & 2 & .3 & 1 & .3 & 7 & 1 & 1 & 3 & 2 & .3 & .2 & .1 \end{bmatrix}$$

The communication delays per data packet in the interconnection network between the six processors are characterized by the matrix C in terms of standard time units per packet:

$$C = \begin{bmatrix} 0 & .1 & .1 & .2 & .2 & .1 \\ .1 & 0 & .4 & .3 & .2 & .1 \\ .1 & .4 & 0 & .2 & .3 & .3 \\ .2 & .3 & .2 & 0 & .3 & .2 \\ .2 & .2 & .3 & .3 & 0 & .1 \\ .1 & .1 & .3 & .2 & .1 & 0 \end{bmatrix}$$

Relative priorities among the 13 tasks are specified thus:

$$W = [13\ 12\ 11\ 8\ 9\ 10\ 7\ 6\ 5\ 4\ 3\ 1\ 2]$$

It should be noted that this priority scheme is entirely arbitrary as is the allocation scheme. The numerical and simulation results shown in Fig. 7 correspond to the probability of completion at time t, $P(X \leq t)$ of the overall job based on three communication scenarios: a) there are no communication costs, b) communication occurs over a high- performance network, and c) communication takes place over a low-performance network. The *MTTC* results along with confidence intervals are given in Table 1. Up to 1000 task graphs were simulated and the time to render averaged results took about 1.69 secs. compared with 125.13 secs. needed by the numerical tool (SPNP) in a Sparc classic workstation. This difference is in part due to the large number of states generated. For the case of the low performance network, SPNP took 2.40 secs. while the simulation process took 0.63 secs [25].

A second application consists in evaluating the task graph shown in Fig. 8. This 20-node task graph describes the LU decomposition algorithm common in the solution of linear systems encountered in many scientific applications. Several schedules for different heuristics were derived in [26]. Two heuristics the Heavy Node First (HFN) and Weighted Length (WL) were examined to determine the corresponding assignment matrices A_{HNF} and A_{WL}, respectively:

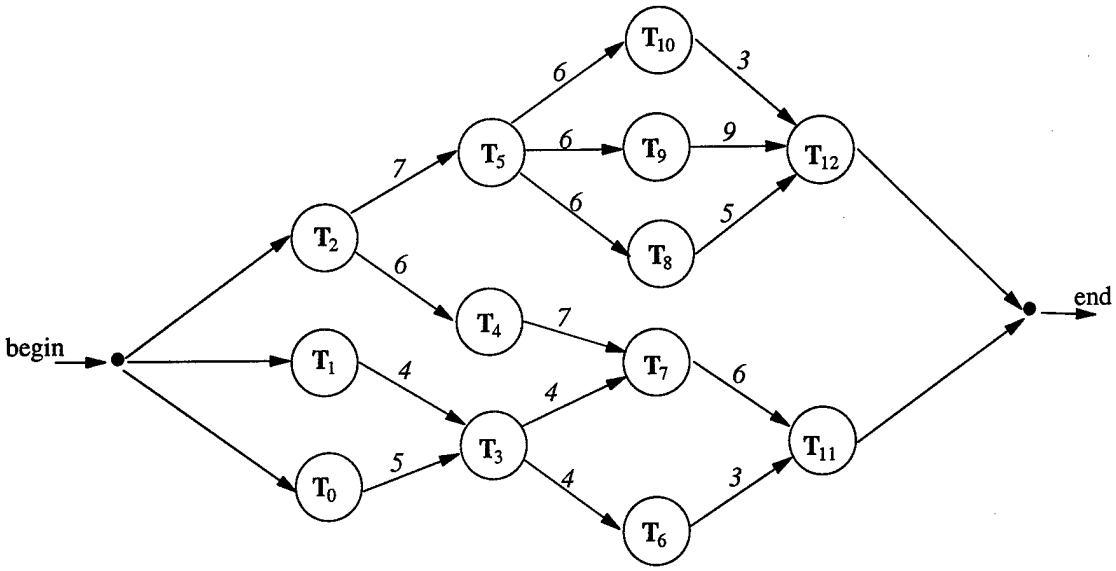


Figure 6. A 13-node complex task graph

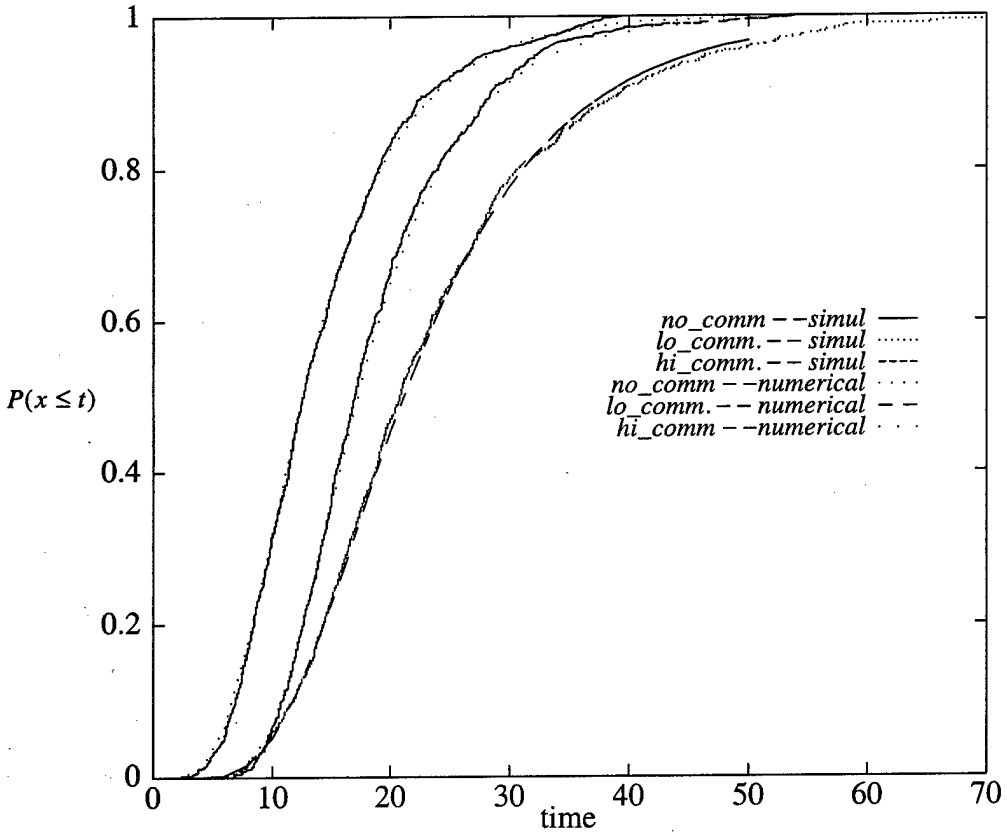


Figure 7. CDF of completion time given static allocation and network type

Table 1. Comparison of MTTC results

Case	Numerical MTTC	Simulation	
		MTTC	99% confidence intervals
High-Performance Network	18.8269	18.5959	17.9854 – 19.2063
Low-Performance Network	23.5204	23.5772	22.6119 – 24.5426
No-communication Costs	14.1999	14.1491	13.5817 – 14.7165

$$A_{HNF}^{T} = \begin{bmatrix} 1\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0\,0\,1\,1\,0\,0\,1\,0\,0 \\ 0\,1\,1\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,1\,0 \\ 0\,0\,0\,1\,0\,0\,0\,1\,1\,1\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1 \end{bmatrix}$$

$$A_{WL}^{T} = \begin{bmatrix} 1\,0\,1\,0\,0\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1 \\ 0\,1\,0\,1\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,1\,1\,0\,0\,1\,0 \\ 0\,0\,0\,0\,1\,0\,1\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0 \end{bmatrix}$$

Both heuristics are based on the execution times (weights) of each task. The HNF heuristic examines the task graph level by level assigning the heaviest nodes first. The WL heuristic assigns control nodes first by associating a rank determined in terms of the length of an exit path, branching factor, number of depending tasks in the path and their weights. For further details see [26]. The schedules reported in the form of Gantt charts were derived assuming the following:

1) The processing units are identical,

2) A communication over processing time ratio very high. Consequently, communication delays are assumed negligible, and

3) Execution times as shown in Fig. 7.

The simulation of these two heuristics under a uniform distribution with zero variance rendered the same total execution time of 96 units. Again examining the schedules the following priority vectors were obtained:

W_{HNF} = [20 19 16 15 14 13 8 9 12 18 7 3 6 11 17 4 2 5 10 1]

W_{WL} = [20 19 17 16 15 11 8 10 14 18 7 3 6 12 13 4 2 5 9 1]

Thus, any instance of heuristics given in the form of Gantt charts such as those derived to compare declustering techniques in [27] can be similarly characterized for the simulation procedure. Fig. 9 shows the plots obtained in the evaluation of SEETF, MFT, LTF, HNF, and WL heuristics under the assumption of exponentially distributed execution times. The priority vectors for the first three schemes were derived as the assignments were made. The results show that HNF and WL perform better as expected. Again 1000 copies of the task graph were used in the simulation.

10. Conclusions

The numerical solution of task graphs based on a GSPN model is limited to execution times that are exponentially distributed. A reliable evaluation of large complex task graphs is not guaranteed as it involves the solution of an underlying very large state space. One way to circumvent this problem is using simulation. The simulation technique discussed relies on a the PN-based topology of a given task graph. Besides naturally capturing the dynamics of a job execution, another advantage in relying on a PN-based topology is that a common model is used for both a numerical and a simulation-based analysis. This is useful in the development of a user interface currently under construction that incorporates both methods of solution.

The simulation tool presented facilitates the analysis and comparison of allocation heuristics. This is illustrated by the evaluation of four heuristics for a particular application. Results are reported to compare the behavior of two types of networks and a comparison is made between simulation results and those obtained using a numerical evaluation. The results of these comparisons validate the simulation tool implemented. It turns our that the simulation algorithm implemented is faster than the numerical solution of the cases reported because of the largeness problem. However, an interface currently under development is required to handle large applications involving thousands of tasks. Also, the tool can be used to explore and determine optimal size of networks in terms of the number of processors to achieve the best performance of a particular application. Since simulation avoids the problem of state explosion present in Markov-based models, a useful extension to this tool must include the analysis of multiple task graphs for which the use of color Petri nets would be more suitable.

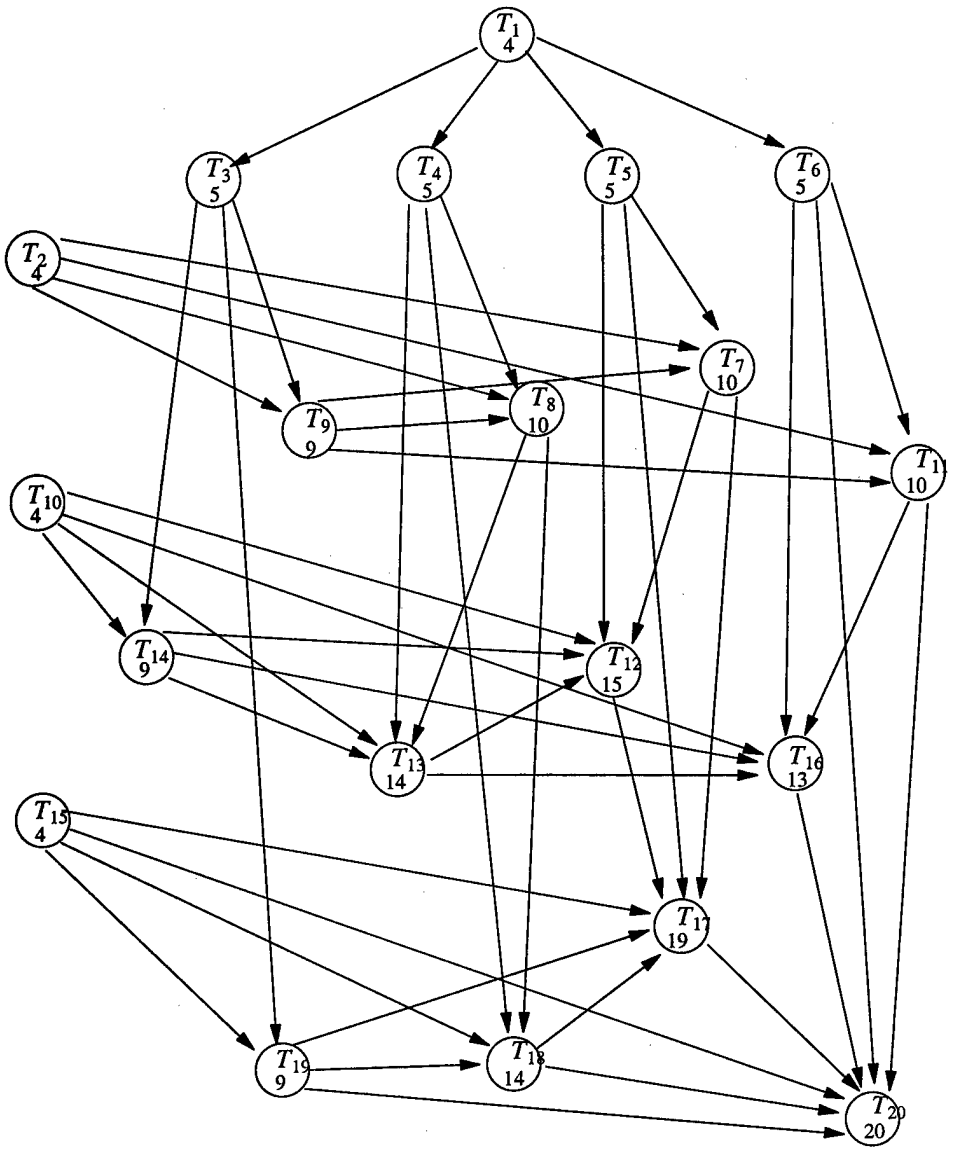


Figure 8. Task Graph for the LU decomposition algorithm

11. Acknowledgements

The authors wish to aknowledge the comments of several anonymous reviewers that greatly improved this paper. This work was partially supported by the National Science Foundation under Grant No. CCR-9520226.

References

[1] R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems,* Kluwer Academic Publishers , 1996.

[2] D. Towsley, C. G. Rommel, and J. A. Stankovic, "Analysis of Fork-Join Program Response Times on Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* Vol. 1, No. 3, July 1990, pp. 286-303.

[3] Y. A. Li and J. K. Antonio, "Estimating the Execution Time Distribution for a Task Graph in a Heterogeneous Computing System," *IEEE Proc. Sixth Heterogeneous Computing Workshop (HCW '97),* Apr. 1997, pp. 172-184.

[4] V. W. Mak and S. F. Lundstrom, "Predicting Performance of Parallel Computations," *IEEE Trans. Parallel and Distributed Systems,* Vol. 1, No. 3, July 1990, pp. 257-270.

[5] A. Thomasian and P. F. Bay, "Analytic Queueing Network Models for Parallel Processing of Task Systems," *IEEE Trans. Comp.,* Vol. C-35, No. 12, Dec. 1986, pp. 1045-1054.

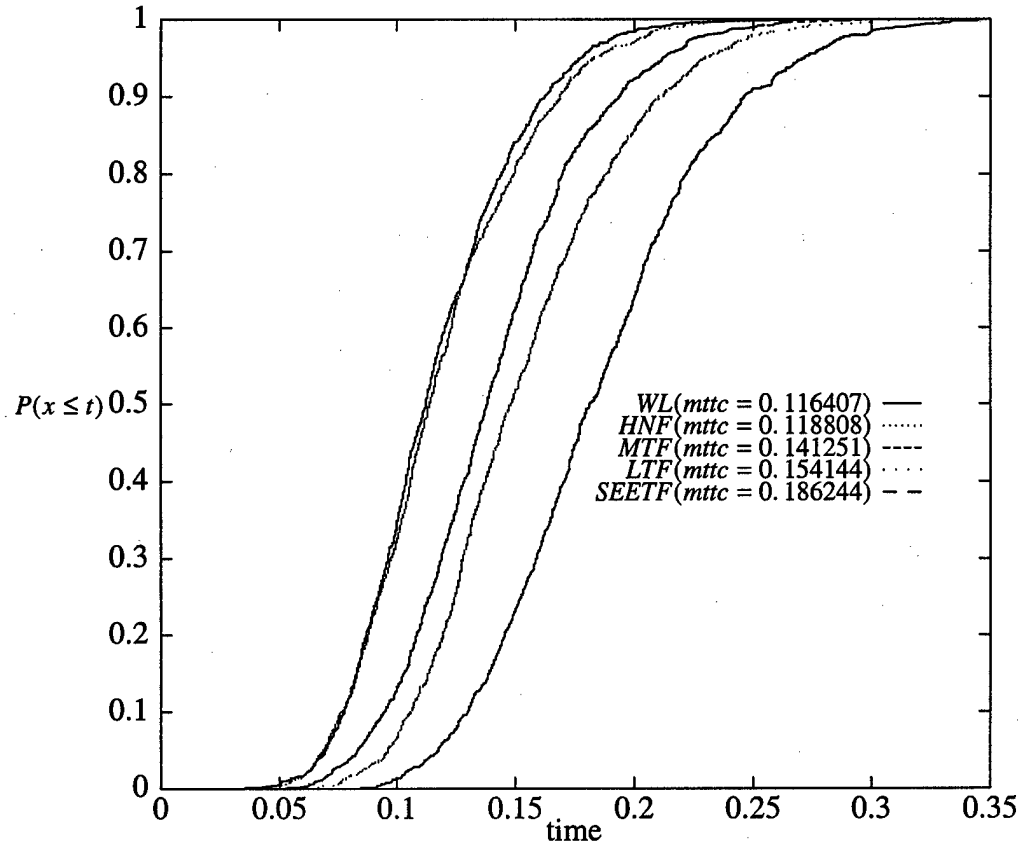


Figure 9. CDF of completion time for the LU decomposition algorithm

[6] D. A. Menasce, D. Saha, S. C. Da Silva Porto, V. A. F. Almeida, and S. K. Tripathi, "Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures," *Parallel and Distributed Computing,* Vol. 28, 1995, pp. 1-18.

[7] N. Lopez-Benitez and K. S. Trivedi, "Performability of Multiprocessor Systems ," *IEEE Trans. Reliability,* Vol. 42 No. 4, Dec. 1993.

[8] N. Lopez-Benitez, "Dependability Modeling and Analysis of Distributed Programs," *IEEE Trans. Software Engineering,* Vol. 20 No. 5, May 1994, pp. 345-352.

[9] Ajmone Marsan. A., Balbo, G. G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stohcastic Petri Nets,* Wiley, Series in Parallel Computing, 1995.

[10] McSpadden Albert R. and Lopez-Benitez Noe, "Stochastic Petri Nets Applied to the Performance Evaluation of Static Task Allocations in Heterogeneous Computing Environments," *Proceedings Heterogeneous Computing Workshop (HCW97)*, Apr. 1997, pp. 185-194.

[11] P. J. Haas and G. S. Shedler, "Stochastic Petri Net Representation of Discrete Event Simulations," *IEEE Trans. Software Engineering,* Vol. 15, No. 4, Apr. 1989, pp. 381-393.

[12] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. IEEE,* Vol. 77, No. 4, Apr. 1989, pp. 541-580.

[13] M.K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Trans. Comp.,* Vol. C-39 No. 9, Sept. 1982, pp. 913-917.

[14] M. A. Marsan, G. Conte, and G. Balbo, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Trans. Computer Systems,* Vol. 2 No.2, May 1984, pp. 93-122.

[15] G. Ciardo, "Toward a Definition of Modeling Power of Stochastic Petri Net Models," *Proceeding Intn'l Workshop on Petri Nets and Performance Models,*

[16] R. A. Sahner and K. S. Trivedi, "Performance and Reliability Analysis Using Directed Acyclic Graphs," *IEEE Trans. Software Engineering,* Vol. SE-13 No. 10, Oct. 1987, pp. 1105-1114.

[17] P. Shroff, D. Watson, N. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," *Proceedings Heterogeneous Computing Workshop 96,* 1996, pp. 98-103.

[18] G. Ciardo, Fricks R. M., J. K. Muppala, and K. S. Trivedi, *SPNP User's Manual* , Version 3.1, Duke University. Dept. of Electrical Engineering, 1993.

[19] M. Ajmone-Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems,* The MIT Press, Cambridge, MA., 1986.

[20] D. Menasce, S. Porto, and Tripathi, s, "Static Heuristic Processor Assignment in Heterogeneous Multiprocessors," *Int'l Journal of High Speed Computing,* Vol. 6,

No.1, 1994, pp. 115-137.

[21] S. C. S. Porto, *Heuristic Scheduling Algorithms for Task Scheduling in Heterogeneous Multiprocessor Architectures,* Master's Thesis, Departamento de Informatica, PUCRIO, 1991.

[22] S. C. S Porto and M. A. Menasce, "Processor Assignment in Heterogeneous Message Passing Parallel Architectures," *Proc. of the HICSS-26 Hawaii Int. Conf. in System Science,* Jan. 1993.

[23] K. C. -Y. Kung, *Concurrency in Parallel Processing Systems,* Ph.D. Dissertation, Dept. of Comp. Sci., University of California, 1984.

[24] C. Anglano, "Performance Modeling of Heterogeneous Distributed Applications," *IEEE MASCOTS'96, Proceedings 4th. Intn'l Workshop,* 1996, pp. 64-68.

[25] Hyon, Ja-Young, *Simulation of Task Graph Systems Using Stochastic Petri Net Models,* MS Thesis, Dept. of Computer Science, Texas Tech University, 1998.

[26] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *J. of Parallel and Distributed Computing* , Vol. 10, 1990, pp. 222-223.

[27] C. G. Sih and E. A. Lee, "Declustering: A New Multiprocesor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 4, No. 6, June 1993, pp. 625-637.

Author Biographies

Noé Lopez-Benitez received the BS degree in Communications and Electronics from the University of Guadalajara, Guadalajara, Mexico. The MS degree in Electrical Engineering from the University of Kentucky, and the PhD in Electrical Engineering from Purdue University in 1989. From 1980 to 1983, he was with the IIE (Electrical Research Institute) in Cuernavaca, Mexico. From 1989 to 1993, he served in the Dept. of Electrical Engineering at Louisiana Tech University. He is now at Texas Tech University in the Department of Computer Science. His research interests include fault-tolerant computing systems, reliability and performance modeling and distributed processing. He is a member of the IEEE, the IEEE Computer Society and the ACM.

Ja-Young Hyon received the BS degree in Statistics from Ewha Women University, Seoul, Korea, in 1994. The BA in Computer Science from Seatle Pacific University in Seattle, WA., and the MS degree in Computer Science at Texas Tech University. Her research interests are in Distributed Computing, Simulation and Statistics.

Communication Modeling of Heterogeneous Networks of Workstations for Performance Characterization of Collective Operations *

Mohammad Banikazemi Jayanthi Sampathkumar Sandeep Prabhu
Dhabaleswar K. Panda P. Sadayappan

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
Email: {banikaze,sampath,sprabhu,panda,saday}@cis.ohio-state.edu

Abstract: *Networks of Workstations (NOW) have become an attractive alternative platform for high performance computing. Due to the commodity nature of workstations and interconnects and due to the multiplicity of vendors and platforms, the NOW environments are being gradually redefined as* Heterogeneous Networks of Workstations *(HNOW). Having an accurate model for the communication in HNOW systems is crucial for design and evaluation of efficient communication layers for such systems. In this paper we present a model for point-to-point communication in HNOW systems and show how it can be used for characterizing the performance of different collective communication operations. In particular, we show how the performance of broadcast, scatter, and gather operations can be modeled and analyzed. We also verify the accuracy of our proposed model by using an experimental HNOW testbed. Furthermore, it is shown how this model can be used for comparing the performance of different collective communication algorithms. We also show how the effect of heterogeneity on the performance of collective communication operations can be predicted.*

1 Introduction

The availability of modern networking technologies [1, 4] and cost-effective commodity computing boxes is shifting the focus of high performance computing systems towards Networks of Workstations (NOW). The NOW systems comprise of clusters of PCs/workstations connected over the Local Area Networks (LAN) and provide an attractive price to performance ratio. Many research projects are currently in progress to provide efficient communication and synchronization for NOW systems. However, most of these projects focus on *homogeneous* NOWs, systems comprising of similar kinds of PCs/workstations connected over a single network architecture. The inherent scalability of the NOW environment combined with the commodity nature of the PCs/workstations and networking equipment is forcing the NOW systems to become *heterogeneous* in nature. The heterogeneity could be due to: 1) the difference in processing and communication speeds of workstations, 2) coexistence of diverse network interconnects, or 3) availability of alternative communication protocols. This adds new challenges to providing fast communication and synchronization on HNOWs while exploiting the heterogeneity.

The need for a portable parallel programming environment has resulted in development of software tools like PVM [22] and standards such as the Message Passing Interface (MPI) [14, 20]. MPI has become a commonly accepted standard to write portable parallel programs using the message-passing paradigm. The MPI standard defines a set of primitives for point-to-point communication. In addition, a rich set of collective operations (such as broadcast, multicast, global reduction, scatter, gather, complete exchange and barrier synchronization) has been defined in the MPI standard. Collective communication and synchronization operations are frequently used in parallel applications [3, 5, 7, 13, 15, 18, 21]. Therefore, it is important that these operations are implemented on a given platform in the most efficient manner. Many research projects are currently focusing on improving the performance of point-to-point [17, 23] and collective [11, 16] communication operations on NOWs. However, none of these studies ad-

*This research is supported in part by NSF Career Award MIP-9502294, NSF Grant CCR-9704512, an Ameritech Faculty Fellowship, and an Ohio Board of Regents Collaborative Research Grant.

0-7695-0107-9/99 $10.00 © 1999 IEEE

dress heterogeneity. The ECO package [12], built on top of PVM, uses pair-wise round-trip latencies to characterize a heterogeneous environment. However, such latency measurements do not show the impact of heterogeneity on communication send/receive overhead (the *fixed* component as well as the *variable* component depending on the message length). A detailed model is necessary to develop optimal algorithms for collective communication operations for a given HNOW system. The existence of such a model also allows to predict/evaluate the impact of an algorithm for a given collective operation on a HNOW system by simple analytical modeling instead of a detailed simulation.

In our preliminary work along this direction [2], we demonstrated that heterogeneity in the speed of workstations can have significant impact on the *fixed* component of communication send/receive overhead. Using this simple model, we demonstrated how near-optimal algorithms for broadcast operations can be developed for HNOW systems. However, this model does not take care of the *variable* component of communication overhead and the transmission component.

In this paper, we take on the above challenges and develop a detailed communication model for collective operations. First we develop a model for point-to-point communication. We present a methodology to determine different components of this model through simple experiments. Next, we show how this model can be used for evaluating the performance of various collective communication operations based on the configuration of the system and the algorithm used for that collective operation. The correctness of this model is validated by comparing the results predicted by this model with the results gathered from our experimental testbed for different system configurations and message sizes. Finally, we illustrate how this model can be used by algorithm designers to select strategies for developing optimal collective communication algorithms and by programmers to study the impact of a HNOW system configuration on the performance of a collective operation.

This paper is organized as follows. The communication model for point-to-point operations is presented in Section 2. The communication model for a set of collective operations and the evaluation of these models are discussed in Section 3. In Section 4, it is shown how the proposed model can be used for comparing different schemes for implementing collective communication operations. It is also shown how we can predict the performance of collective operations with changes in system configuration. We conclude the paper with future research directions.

2 Modeling Point-to-point Communication

For the characterization of collective communication operations on heterogeneous networks, the knowledge of send and receive costs on the various nodes is imperative. In most implementations of MPI, such as MPICH [8], collective communication is implemented using a series of point-to-point messages. Hence, from the characterization of point-to-point communication on the various type of nodes in a heterogeneous network, the cost of any collective communication operation can be estimated. This section explains the method adopted to determine the send and receive costs as a function of the message size for point-to-point communication in a heterogeneous network.

2.1 Characterization of Communication Components

The cost of a point-to-point message transfer consists of three components, namely, the send overhead, transmission cost and the receive overhead. These components comprise of a message size dependent factor and a constant factor. Thus, the one-way time for a single point-to-point message between two nodes can be expressed as:

$$T_{ptp} = O_{send} + O_{trans} + O_{receive} \quad (1)$$

$$O_{send} = S_c^{sender} + S_m^{sender} \cdot m \quad (2)$$

$$O_{trans} = X_c + X_m \cdot m \quad (3)$$

$$O_{receive} = R_c^{receiver} + R_m^{receiver} \cdot m \quad (4)$$

where m is the message size (in bytes). The components S_c, X_c, and R_c are the constant parts of the send, transmission and receive costs respectively. The components $S_m \cdot m$, $X_m \cdot m$, and $R_m \cdot m$ are the message dependent parts.

To obtain an empirical characterization of point-to-point communication, each term in these equations needs to be measured. In order to measure these terms we need two sets of experiments: *ping-pong experiment* and *consecutive sends experiment*. The time for completing a point-to-point message between a pair of nodes can be measured using a ping-pong experiment. In this experiment, one of the nodes performs a send followed by a receive while the other does a receive followed by a send. The round-trip time is determined by averaging over multiple such iterations. If the nodes involved are identical, the time for a point-to-point message transfer is equal to half the round-trip time. For small messages, the message size dependent cost can be ignored compared to the constant costs. Therefore,

$$T_{ptp,small} = \frac{1}{2} T_{ping-pong,small} \approx S_c^{sender} + R_c^{receiver} \quad (5)$$

The send component of point-to-point messages (S_c) in the above equation can be obtained by measuring the time for a small number of consecutive sends from one of the nodes and averaging the time over the number of sends.

Using the measured value of S_c and the measured value of $T_{ptp,small}$, R_c can be obtained from Equation 5. The consecutive sends experiment can be used repeatedly for varying message sizes to calculate the send overhead for those message sizes. The linear fit of the send overheads can be plotted as a function of message size. The component S_m is equal to the slope of the straight line fitted on this plot. The transmission cost (O_{trans}) can be determined from the configuration of the underlying network. Then, R_m can be measured by taking the difference of the T_{ptp} obtained from the ping-pong experiment and the sum of all other components in Equation 1.

2.2 Measurement of Communication Components

In this section we describe the testbed used to verify our proposed model and present the measured values of different components of point-to-point communication.

We used two types of personal computers in our testbed. The first group of nodes were Pentium Pro 200MHz PCs with 128MB memory and 16KB/256KB L1/L2 cache. These machines are referred to as *slow* nodes in the rest of the paper. The second group of nodes were Pentium II 300MHz PCs with 128MB memory and 32KB/256KB L1/L2 cache. We refer to these machines as *fast* nodes. All the nodes were connected to a single Fast Ethernet switch with a peak bandwidth of $100Mbits/sec$ and a $8\mu sec/port$ latency. Thus, the transmission cost ($X_m \cdot m + X_c$) for our testbed can be expressed as $(m/12.5 + 2 * 8)\mu sec$. The factor of two in this expression is used to reflect the effect of latencies for both input and output switch ports in the one-way communication.

We performed the experiments described in Section 2.1 for both slow and fast nodes. The components corresponding to the slow and fast nodes are shown in Table 1.

Table 1. Send and Receive parameters.

Type	S_c (μsec)	S_m	R_c (μsec)	R_m
slow	90.0	0.18	140.0	0.08
fast	60.0	0.05	110.0	0.03

Observation 1 *The send overhead and receive overhead can be different in a heterogeneous environment. Using the same value for both overheads or just considering the send overhead may result in inaccurate models.*

We also verified these values by measuring the one way latency of messages between different types of nodes. Table 2 illustrates the latency (of zero-byte messages) between fast and slow nodes. The experimental results corresponding to one-way latencies agree with those calculated from Equations 1 through 4. It should be noted that the same set of experiments can be used for systems with multiple types of computing nodes. In general, for a system with n different types of computers, the experiments should be performed n times (once for each type).

Table 2. One-way latency (microsec) between different types of nodes in our testbed.

Sender Type	Receiver Type	Time (μsec)
Fast	Fast	170
Fast	Slow	200
Slow	Fast	200
Slow	Slow	230

In heterogeneous environments, there is a possibility of having different types of machines at the sending and receiving sides of a point-to-point communication. Thus, using a simple model for communication cost such as $T_{ptp} = O_{constant} + O_{per-byte} \cdot m$ with the same $O_{constant}$ and $O_{per-byte}$ for all pairs of nodes will not be sufficient.

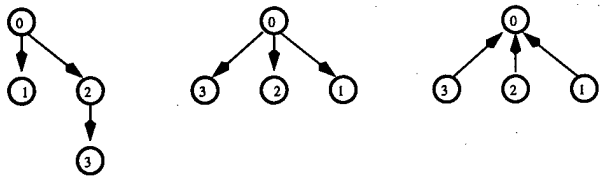
3 Modeling Collective Communication Operations

The MPI standard provides a rich set of collective communication operations. Different implementations of MPI use different algorithms for implementing these operations. However, most of the implementations, specially those used for NOW environments, implement the different collective operations on top of point-to-point operations. Different types of trees (e.g. binomial trees, sequential trees, and k-trees) can be used for implementing these operations [6, 9, 10, 19]. We can classify the MPI collective communication operations into three major categories: *one-to-many* (such as MPI_Bcast and MPI_Scatter), *many-to-one* (such as MPI_Gather), and *many-to-many* (such as MPI_Allgather and MPI_Alltoall). We present the analysis for some of the representative operations.

In the following sections, we provide analytical models for the broadcast, scatter, and gather operations as they are implemented in MPICH. We also verify the accuracy of our analytical models by comparing the estimated times with measured times on the experimental testbed. It should be noted that our model does not consider the effect of contention and is applicable only to fully-connected systems.

3.1 Broadcast and Multicast

Binomial trees are used by MPICH for implementing broadcast and multicast. Figure 1a illustrates how broadcast is performed on a four-node system. The completion



(a) Broadcast Tree (b) Scatter Tree (c) Gather Tree

Figure 1. Trees used in MPICH for the broadcast, scatter, and gather operations. The numbers inside the tree nodes indicate the rank of the computing nodes assigned to those tree nodes.

time of broadcast on a n-node system can be modeled by using the following expression:

$$T_{broadcast} = max\{T^0_{recv}, T^1_{recv}, \cdots, T^{n-1}_{recv}\} \quad (6)$$

where T^i_{recv} is the time when node i receives the entire message. The value of T_{recv} for the root of broadcast is obviously zero, and T_{recv} for all other nodes can be calculated from the following recurrence:

$$\begin{aligned} T^i_{recv} &= T^{parent(i)}_{recv} + \\ & childrank(parent(i), i) \cdot \\ & \quad (S^{parent(i)}_c + S^{parent(i)}_m \cdot msg\_size) + \\ & X_m \cdot msg\_size + X_c + \\ & R^i_m \cdot msg\_size + R^i_c \end{aligned} \quad (7)$$

where $parent(i)$ indicates the parent of node i in the broadcast tree and $childrank(parent(i), i)$ is the order, among its siblings, in which node i receives the message from its parent.

This model can be used to estimate the completion time of a broadcast operation on a given heterogeneous system. In order to verify the accuracy of this model, we also measured the completion time of MPI_Bcast on our experimental testbed. The procedure in Figure 2 was used for measuring the completion time of MPI_Bcast (or other MPI collective operations). In order to minimize the effect of external factors, this procedure was executed several times and the minimum of the measured times is reported.

The comparison between the measured and estimated completion times of broadcast on a four-node system with four different configurations is shown in Figure 3. It can be observed that the estimated times using our analytical models are very close to the measured times. It is also interesting to note that the completion time of broadcast for a system with two fast nodes and two slow nodes varies with the order of the nodes in the binomial tree (configurations B and D).

```
MPI_Barrier
get start_time
for (i=0; i < ITER; i++) {
   MPI_Barrier
}
get end_time
barrier_time= (end_time - start_time) / ITER
MPI_Barrier
get start_time
for (i=0; i < ITER; i++) {
   MPI_Bcast /* or any other collective operation */
   MPI_Barrier
}
get end_time
local_time= (end_time - start_time) / ITER
global_time= reduce(local_time, MAXIMUM)
broadcast_time= global_time-barrier_time
```

Figure 2. Outline of the procedure used for measuring the completion time of broadcast and other collective operations.

Observation 2 *The completion time for a given broadcast tree depends not only on the type of participating nodes, but also on how these nodes are assigned to the broadcast tree nodes.*

3.2 Scatter

Scatter is another one-to-many operation defined in the MPI standard. Sequential trees are used to implement scatter in MPICH[1]. Figure 1b illustrates how scatter is implemented on a four-node system. The completion time of scatter on an n-node system can be modeled as follows:

$$T_{scatter} = max\{T^0_{recv}, T^1_{recv}, \cdots, T^{n-1}_{recv}\} \quad (8)$$

$$\begin{aligned} T^i_{recv} &= childrank(root, i) \cdot \\ & \quad (S^{parent(i)}_c + S^{parent(i)}_m \cdot msg\_size) + \\ & X_m \cdot msg\_size + X_c + \\ & R^i_m \cdot msg\_size + R^i_c \end{aligned} \quad (9)$$

Figure 4 compares the estimated completion times (based on Equation 8) of the scatter operation with those measured on the experimental testbed with four nodes and different configurations. It can be observed that the estimated times (using our analytical models) are very close to the measured times.

[1]It should be noted that in the scatter operation, as implemented in MPICH, a message is sent from the root to itself through point-to-point communication. Since the time for this operation is small in comparison with the total completion time of scatter we ignore it in our analysis. We use the same approach when we model the gather operation.

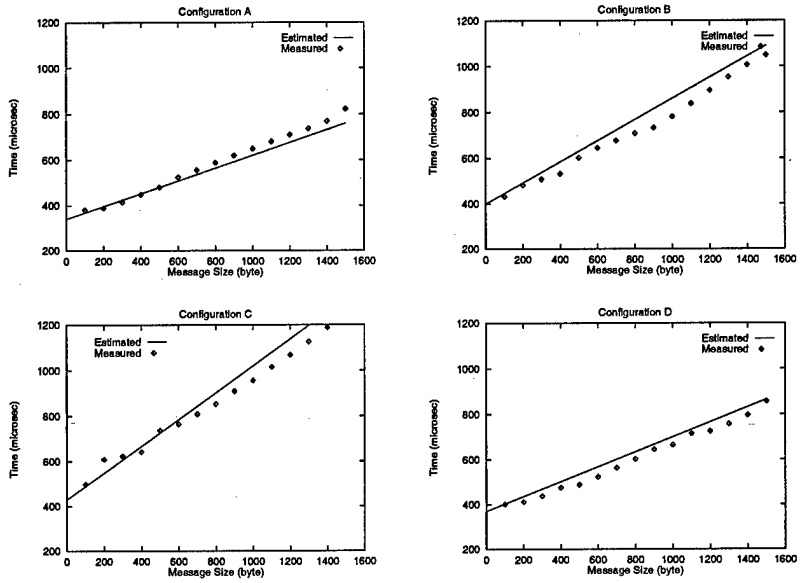


Figure 3. Estimated and measured completion times for the broadcast operation on four nodes with four different configurations. The types of the participating nodes from node 0 to node 3 (in MPI ranking) are: A: fast, fast, fast, fast; B: fast, fast, slow, slow; C: slow, fast, slow, fast; D: fast, slow, fast, slow.

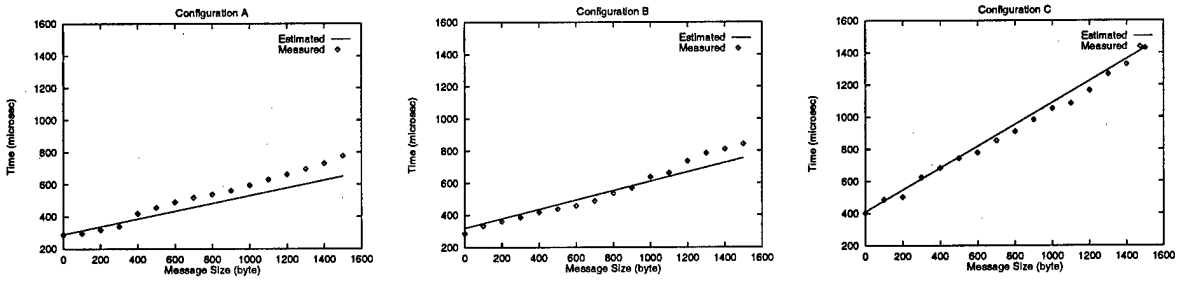


Figure 4. Estimated and measured completion times for the scatter operation on four nodes with three different configurations. The types of the participating nodes from node 0 to node 3 (in MPI ranking) are: A: fast, fast, fast, fast; B: fast, fast, slow, slow; C: slow, fast, slow, fast.

3.3 Gather

Gather is a many-to-one collective operation implemented in MPICH by using reverse sequential trees (Fig. 1c). The completion time of this operation can be modeled by using the following expression:

$$T_{gather} = T^{n-1}_{receive} \quad (10)$$

$$T^i_{receive} = max\{T^{i-1}_{receive}, T^i_{arrive}\} + R^{root}_m \cdot msg\_size + R^{root}_c \quad (11)$$

$$T^i_{arrive} = (\ The\ ith\ smallest\ element\ of\ \{S^j_c + S^j_m \cdot msg\_size\}\ for\ 1 < j < n-1\ ) + i \cdot (X_m \cdot msg\_size + X_c) \quad (12)$$

where $T^i_{receive}$ is the time by which the messages from i nodes (out of the the $n-1$ sender nodes) have been received at the root node. T^i_{arrive} is the time when the ith message has arrived at the root node.

The estimated completion times using our models and the measured completion times for a four-node system with different configurations are shown in Figure 5. It can be seen that the estimated times are close to the measured times.

4 Applying the Communication Model

In this section, we explain how the models developed in the previous sections can be used for evaluating different collective communication algorithms. We also discuss how these models can be used to characterize the effect of the heterogeneous configuration (the number of nodes from different types). Without loss of generality, we consider systems with two different types of computing nodes.

4.1 Choice of Algorithms

Collective communication operations can be implemented by using different algorithms (which use different types of trees). For instance, MPICH and many other communication libraries use the binomial trees for broadcast. However, it has been shown that binomial trees are not the best choice for all systems [2]. Therefore, in order to find the best scheme for a given collective operation, it is important to compare the performance of different schemes. Our proposed communication model can be used to evaluate the performance of these algorithms analytically.

Consider an 8-node system (four fast nodes and four slow nodes with characteristics described in Section 2.2) and three different trees (as shown in Fig. 6) for the broadcast operation. The estimated completion times of broadcast on this system using different tree structures are shown in Figure 7a. It can be seen that for the given configuration, the binomial tree performs worse than the other trees. For messages up to $4K$ bytes in size, the hierarchical tree performs better than others. For messages larger than $4K$ bytes, the sequential tree has the best performance. Figure 7b shows the results for a 16-node system. It can be seen that the hierarchical algorithm outperforms the other two algorithms for the 16-node system.

The models presented in this paper can be used to evaluate the performance of different collective communication operations in heterogeneous environments. These models can be used for comparing the performance of different algorithms for different collective operations and identifying the most efficient algorithms.

4.2 Effect of Configuration on Performance

The proposed communication models can also be used for predicting the effect of configurations in HNOW systems. For instance, consider a system with a set of slow and fast nodes. We are interested in knowing how the performance of collective communication operations varies based on the number of fast/slow nodes in the system. Figure 8 illustrates the estimated completion times of the gather operation for varying number of fast nodes in systems with 8 and 16 nodes, respectively. (The root is always considered to be a fast node.) It can be observed that having more than four fast nodes in these systems does not improve the performance of this operation significantly. The models presented in this paper can be used to evaluate the performance of other collective communication operations and other system configurations.

5 Conclusions and Future Work

In this paper, we have proposed a new model for estimating the cost of point-to-point and different collective operations in the emerging HNOW systems. We have verified the validity of our models by using an experimental heterogeneous testbed. In addition, we have shown how this model can be used to compare different algorithms for different collective operations. We have also shown that this model can be used to predict the effect of having different types of computing nodes on the performance of collective operations.

We plan to evaluate our model by using other promising networking technologies (such as Myrinet and ATM) and underlying point-to-point communication layers (such as FM and U-Net). We also plan to extend our model for systems with heterogeneous networking technologies. We are exploring how this model can be used to predict the execution time of parallel applications on heterogeneous systems. By doing so, we will be able to consider the effect

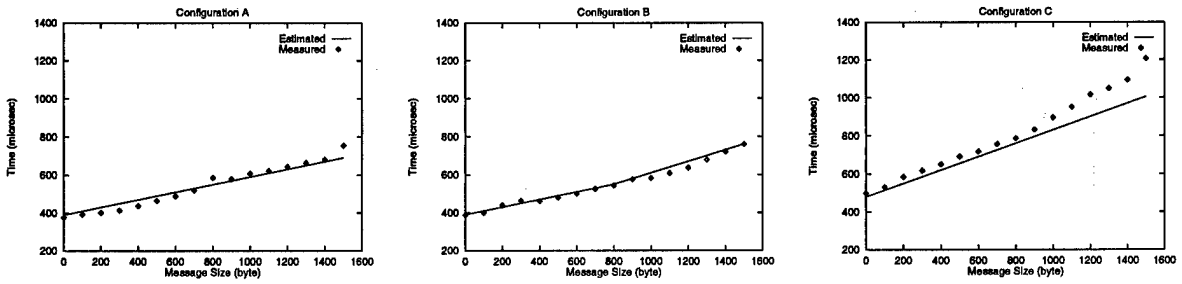
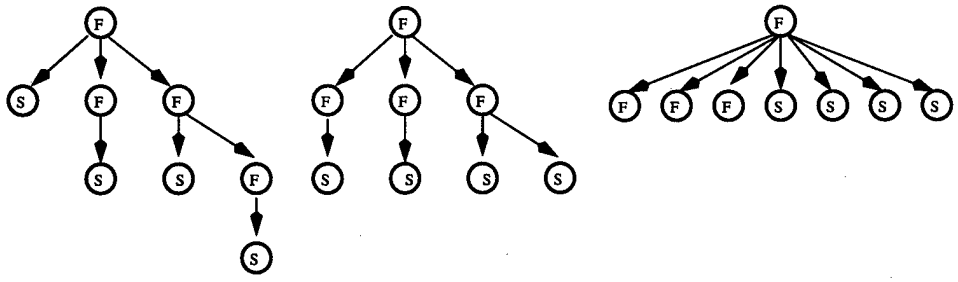


Figure 5. Estimated and measured completion times for the gather operation on four nodes with three different configurations. The types of the participating nodes from node 0 to node 3 (in MPI ranking) are: A: fast, fast, fast, fast; B: fast, fast, slow, slow; C: slow, fast, slow, fast.



(a) Binomial Tree (b) Hierarchical Tree (c) Sequential Tree

Figure 6. Three different possible trees for implementing broadcast in a HNOW system with eight nodes (F = fast node, S = slow node).

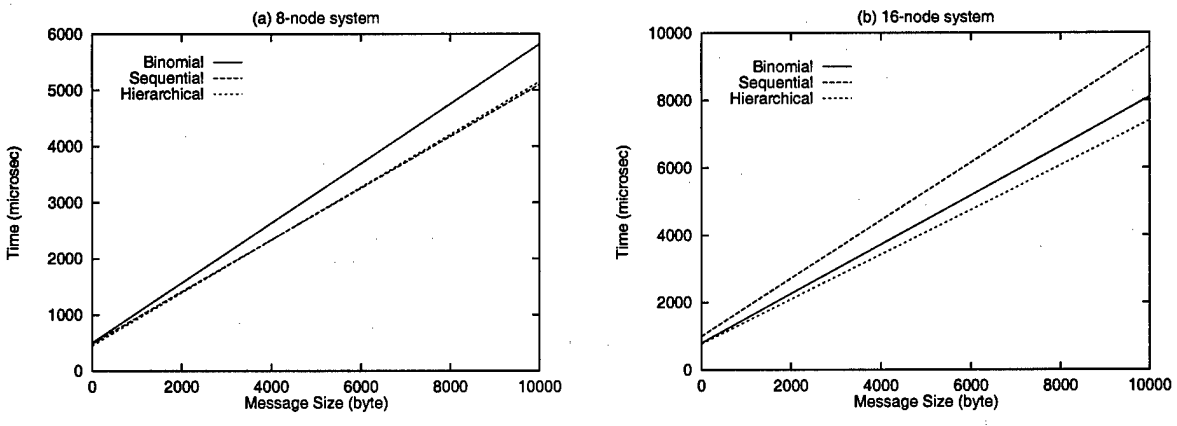


Figure 7. Comparing the performance of three different broadcast algorithms (based on the respective tree structures of Figure 6) on 8-node and 16-node heterogeneous systems.

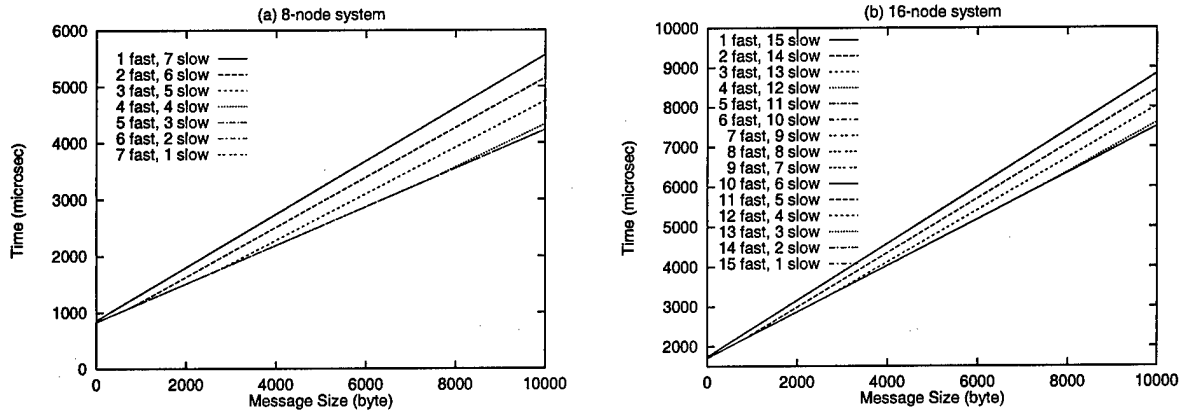


Figure 8. Comparison between the completion times of the gather operation on 8-node and 16-node systems with different number of fast and slow nodes.

of communication time in overall execution time of an application more accurately and based on that come up with better load balancing schemes.

References

[1] ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[2] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient Collective Communication on Heterogeneous Networks of Workstations. In *International Conference on Parallel Processing*, pages 460–467, 1998.

[3] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor Collective Communication Library (Intercom). In *Scalable High Performance Computing Conference*, pages 357–364, 1994.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovik, and Wen-King Su. Myrinet — A Gigabit-per-Second Local-Area Network. *IEEE MICRO*, 15(1):29–36, February 1995.

[5] R. V. Boppana, S. Chalasani, and C. S. Raghavendra. On Multicast Wormhole Routing in Multicomputer Networks. In *Symposium on Parallel and Distributed Processing*, pages 722–729, 1994.

[6] J. Bruck, R. Cypher, P. Elustando, A. Ho, C.T. Ho, V. Bala, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. In *Proceedings of the International Parallel Processing Symposium*, 1994.

[7] D. Dai and D. K. Panda. Reducing Cache Invalidation Overheads in Wormhole DSMs Using Multidestination Message Passing. In *International Conference on Parallel Processing*, pages I:138–145, Chicago, IL, Aug 1996.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[9] R. Kesavan, K. Bondalapati, and D. K. Panda. Multicast on Irregular Switch-based Networks with Wormhole Routing. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-3)*, pages 48–57, February 1997.

[10] R. Kesavan and D. K. Panda. Multiple Multicast with Minimized Node Contention on Wormhole k-ary n-cube Networks. *IEEE Transactions on Parallel and Distributed Systems*. in press.

[11] M. Lin, J. Hsieh, D. H. C. Du, J. P. Thomas, and J. A. MacDonald. Distributed Network Computing over Local ATM Networks. *IEEE Journal on Selected Areas in Communications*, 13(4), May 1995.

[12] B. Lowekamp and A. Beguelin. ECO: Efficient Collective Operations for Communication on Heterogeneous Networks. In *Int'l Parallel Processing Symposium*, pages 399–405, 1996.

[13] P. K. McKinley and D. F. Robinson. Collective Communication in Wormhole-Routed Massively Parallel Computers. *IEEE Computer*, pages 39–50, Dec 1995.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[15] P. Mitra, D. G. Payne, et al. Fast Collective Communication Libraries, Please.

[16] N. Nupairoj and L. M. Ni. Performance Evaluation of Some MPI Implementations on Workstation Clusters. In *Proceedings of the SPLC Conference*, 1994.

[17] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[18] D. K. Panda. Issues in Designing Efficient and Practical Algorithms for Collective Communication in Wormhole-Routed Systems. In *ICPP Workshop on Challenges for Parallel Processing*, pages 8–15, 1995.

[19] R. Sivaram, D. K. Panda, and C. B. Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October 1998.

[20] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996. Chapter 4 of this book deals with collective communications.

[21] C. B. Stunkel, R. Sivaram, and D. K. Panda. Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and their Impact. In *Proceedings of the 24th IEEE/ACM Annual International Symposium on Computer Architecture (ISCA-24)*, pages 50–61, June 1997.

[22] V. S. Sunderam. PVM: A Framework for Parallel and Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

Biographies

Mohammad Banikazemi is a Ph.D. student in the Department of Computer and Information Science at The Ohio State University. His research interests include network-based computing, heterogeneous computing, and interprocessor communication. He received an M.S. degree in Electrical Engineering from The Ohio State University in 1996, and a B.S. degree in Electrical Engineering from Isfahan University of Technology, Isfahan, Iran.

Jayanthi Sampathkumar is a graduate student at The Ohio State University. Her research is currently focused on parallel computing. She received a B.Tech. degree form Institute of Technology, Banara Hindu University, India.

Sandeep Prabhu is currently working at Microsoft Corporation. He received a MS degree in computer and information science from The Ohio State University and a B.Tech. in Computer Engineering from Indian Institute of Technology, Bombay, India.

Dhabaleswar K. Panda is an Associate Professor in the Department of Computer and Information Science, The Ohio State University, Columbus, USA. His research interests include parallel computer architecture, wormhole-routing, interprocessor communication, collective communication, network-based computing, and high-performance computing. He has published over 60 papers in major journals and international conferences related to these research areas.

Dr. Panda has served on Program Committees and Organizing Committees of several parallel processing conferences. He was a Program Co-Chair of the 1997 and 1998 Workshops on Communication and Architectural Support for Network-Based Parallel Computing and a co-guest editor for special issue volumes of the *Journal of Parallel and Distributed Computing* on "Workstation Clusters and Network-based Computing". Currently, he is serving as a Program Co-Chair of the 1999 International Conference on Parallel Processing, an Associate Editor of the *IEEE Transactions on Parallel and Distributed Computing*, an IEEE Distinguished Visitor Speaker, and an IEEE Chapters Tutorials Program Speaker. Dr. Panda is a recipient of the NSF Faculty Early CAREER Development Award in 1995, the Lumley Research Award at Ohio State University in 1997, and an Ameritech Faculty Fellow Award in 1998. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

P. Sadayappan is a Professor of Computer and Information Science at The Ohio State University. He obtained a B. Tech. in Electrical Engineering from I.I.T. Madras, India, and an M.S. and Ph. D. in Electrical Engineering from S.U.N.Y at Stony Brook. His research interests include network-based computing and high-performance scientific computing.

Session IV

Task Assignment and Scheduling

Chair

Fusun Özgüner
The Ohio State University

Multiple Cost Optimization for Task Assignment in Heterogeneous Computing Systems Using Learning Automata

Raju D. Venkataramana
Dept. of Computer Science and Engineering
University Of South Florida
Tampa, FL 33620, USA
venkatar@csee.usf.edu

N. Ranganathan
Dept. of Electrical and Computer Engineering
University of Texas at El Paso
El Paso, TX 79968, USA
ranganat@ece.utep.edu

Abstract

A framework for task assignment in heterogeneous computing systems is presented in this work. The framework is based on a learning automata model. The proposed model can be used for dynamic task assignment and scheduling and can adapt itself to changes in the hardware or network environment. The important feature of the scheme is that it can work on multiple cost criteria, optimizing each criterion individually. The cost criterion could be a general metric like minimizing the total execution time, or an application specific metric defined by the user. The application task is modeled as a task flow graph(TFG), and the network of machines as a processor graph(PG). The automata model is constructed by associating every task in the TFG with a variable structure learning automaton [1]. The actions of each automaton correspond to the nodes in the PG. The reinforcement scheme of the automaton considered here is a linear scheme. Different heursitic techniques that guide the automata model to the optimal solution are presented. These heuristics are evaluated with respect to different cost metrics.

Key words: *Task assignment, variable structure learning automata, multiple cost criteria, stochastic optimization, task flow graph and processor graph.*

1. Introduction

Heterogeneous computing(HC) [2, 8, 15], is the tuned use of diverse processing hardware to meet distinct computational needs. A HC environment consists of a heterogeneous suite of machines and high speed interconnections, which can be used effectively to execute different portions of an application. An application is usually represented as a task flow graph (TFG), which is a directed acyclic graph showing the data dependencies between the various tasks of the application. An important problem in this domain is the task assignment problem, which corresponds to the assignment of tasks to the machines in the HC suite such that a specified cost criterion is optimized. It is well known that the assignment problem in general is NP-complete [6].

In the literature, a variety of mathematical formulations have been developed for the task assignment problem. These formulations are collectively called *selection theory* and try to choose the appropriate machine for each subtask of the TFG [6, 7, 9, 10, 4]. Approaches to the problem based on graph theoretic techniques [5, 4], simulated annealing [14], exhaustive state space search [16], and genetic techniques [12, 14, 11] have been proposed. In this work, the authors propose a new learning automata model for the assignment problem, which is based on a variable-structure learning automaton [1]. In [3], a stochastic learning automaton model was proposed for scheduling in distributed systems. The systems were assumed to be homogeneous and had a large number of automaton states, 2^M for an M-node network. The model proposed here, associates an automaton with every task in the TFG, and hence there are only M states for an M-node network. The different reinforcement schemes of the automata define the behavior of the assignment algorithm. The key feature of the proposed model is its ability to optimize multiple cost metrics. In other works multimetric optimization is achieved by optimizing the weighted sum of the various metrics. In the proposed model, the cost metrics are optimized independent of each other subject to the weight associated with it. The cost criterion itself could be a general metric like minimizing the total execution time, or it could be defined specifically to suit the needs of an application.

The paper is organized as follows. Section 2 introduces the framework, followed by a description of the HC system model and the cost criteria in section 3. A construction of the automata model and the heuristic techniques proposed

for the model are presented in the next section. The subject of section 5 is the performance analysis of these techniques. The last section provides the conclusions and scope for future work.

2. Model of the Framework

This section presents a general model of the proposed framework for task assignment in the HC system. Figure 1 depicts a schematic representation of the framework. The environment consists of the heterogeneous suite of machines which will be used to execute the application. The scheduling system consists of the proposed learning automata model and a model of the application and HC system. These models are used by the scheduler to assign the subtasks to the different machines.

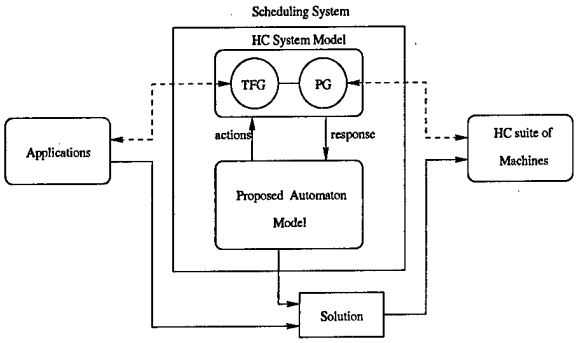


Figure 1. Model of the Proposed Framework

The HC system environment consists of a heterogeneous suite of machines interconnected in a specific topology. These machines could include sequential processors, MIMD systems, SIMD systems, or special purpose architectures and DSP processors. All these various systems are assumed to be interconnected by a high speed interconnection network. The application task is assumed to be represented as a task flow graph(TFG). The TFG is a directed acyclic graph which shows the dependencies of the subtasks. The interconnection of the machines in the HC system is represented similarily by means of a processor graph(PG). The objective of the scheduling system is to assign the subtasks of the TFG to the processors in the PG, so that the defined set of cost criteria are optimized. This is achieved by means of a automata model proposed in this paper. The model is built on a variable structure learning automaton and uses an iterative algorithm. The external environment for this automaton model is the system model formed by the TFG and the PG. Once the scheduling system determines on which processors the various tasks are to be executed, the actual execution on the HC system can take place.

The advantages of the proposed framework can be seen quite clearly. Since the actual task assignment and scheduling is based on a model of the application(TFG) and the machines of the HC system(PG), the framework can be used for dynamic task assignment and scheduling. The proposed learning automaton model optimizes the set of cost criteria based on a given PG. Hence, whenever there is a change in the machine or network configuration, the model will generate a solution that is optimal to the transformed system configuration. The important feature of the proposed framework, is the incorporation of multiple cost criteria, all of which are optimized simultaneously. The following sections explain the construction of the framework and highlight its advantages.

3. HC System Model

This section deals with the modeling of the application and the hardware configuration of the HC system. Certain assumptions that have been made are introduced first. The later part of the section discusses how the different cost metrics for the system can be defined. The task assignment algorithm will generate a solution that tries to optimize these cost criteria.

3.1. Assumptions

1. The application program is assumed to be decomposed into multiple tasks. The data dependencies between the tasks are given by a Task Flow Graph(TFG), which is an acyclic directed graph.
2. The HC system is assumed to consist of a set of heterogeneous machines, which communicate by means of an underlying interconnection network. This is represented by a Processor graph(PG).
3. The expected execution time of the tasks on each machine in the HC suite is known a priori. These execution times can be obtained by task profiling and analytical benchmarking techniques.
4. The cost of communicating a single unit of data between any two machines is also known a priori. If any pair of machines in the HC suite cannot communicate with each other, then the cost of communication between them is assumed to be ∞.

3.2. Cost Metric

The TFG contains a set of tasks S. s_i is the ith task in S. The set of machines in the HC environment is given by M, where m_j is the jth machine. C denotes the set of cost metrics defined for the application. The assignment problem then corresponds to a mapping π from the set S to the set M, such that the metrics in C are optimized.

$S = \{s_i, 0 \leq i < |S|\}$
$M = \{m_j, 0 \leq j < |M|\}$
$C = \{c_k, 0 \leq k < |C|\}$ where c_k is a specific cost metric
$\pi : S \rightarrow M$

A solution vector $X(n)$ of size $|S|$ x $|M|$, gives an instance of the mapping π at iteration n. Each element of the vector, $x_i(n)$ indicates the machine to which that particular task is assigned to at iteration n. The cost of executing a task on a machine, the communication time between the machines, the number of data units exchanged between tasks and the amount of power consumed by executing a task on a machine, are all given by the following matrices.

$X(n) \rightarrow$ Solution Vector of order $|S|$ x $|M|$
$EX \rightarrow$ Execution cost matrix of order $|S|$ x $|M|$
$CC \rightarrow$ Communication cost matrix of order $|M|$ x $|M|$
$DX \rightarrow$ Data exchange matrix of order $|S|$ x $|S|$
$PW \rightarrow$ Power cost matrix of order $|S|$ x $|M|$

The next part in formulating the problem involves the actual definition of the cost metrics. There are a number of such metrics that can be defined by the user based on the application. For instance, the user might want to minimize the total execution time, or minimize the maximum loaded processor, or minimize the amount of power consumed. There may also be other metrics that are specific to a particular application. These may be defined with the help of the matrices defined above. Each of these metrics is defined at an iteration n as $c_k(n)$. A couple of these metrics are defined here:

Minimize the load on the maximum loaded processor:

Let the total load on a particular machine say m_j at iteration n be represented as $l_j(n)$. Therefore

$$l_j(n) = \sum_{i=0,x_i(n)=j}^{|S|-1} ex_{ix_i(n)} + \sum_{i=0,x_i(n)=j}^{|S|-2} \sum_{k=i+1}^{|S|-1} dx_{ik} * cc_{x_i(n)x_k(n)}$$

In the equation for $l_j(n)$, the first part represents the total computation cost on the processor m_j and the second part corresponds to the total communication cost incurred by the processor at iteration n. The load on every processor needs to be computed, in order to identify the heaviest loaded processor. The optimal assignment for the problem would be that which results in minimizing the load on the heaviest loaded processor. Therefore, the cost metric $c_1(n)$ would be:

$$c_1(n) = Max(\, l_j(n)\,) \qquad \text{for } j = 1 \text{ to } |M| - 1$$

Minimize the total execution time:

Let $cp(n)$ represent the total computation time for a particular assignment and $cc(n)$ represent the total communication time at a particular iteration. Hence:

$$cp(n) = \sum_{i=0}^{|S|-1} ex_{ix_i(n)}$$
$$cc(n) = \sum_{i=0}^{|S|-2} \sum_{k=i+1}^{|S|-1} dx_{ik} * cc_{x_i(n)x_k(n)}$$

The total execution time is the sum of the total computation and communication time for the task assignment at an iteration. Therefore for this cost metric, $c_2(n)$ would be:

$$c_2(n) = cp(n) + cc(n)$$

Minimize the power consumed:

The cost metric for this formulation, $c_3(n)$ can be represented as:

$$c_3(n) = \sum_{i=0}^{|S|-1} pw_{ix_i(n)}$$

The task assignment algorithm, would now try to minimize each of the $c_k(n)$ $(k = 1 \text{ to } |C|))$ metrics over n iterations until the absolute minimum is reached.

4. Proposed Learning Automata Model

In the proposed framework, a learning automata model is used to determine an optimal task assignment for the HC system. This model is based on a variable structure learning automaton explained in [1]. The section begins with an introduction to this automaton model. The mathematical formulation and advantages of using the model are discussed. The construciton of such an automata model for the purpose of task assignment in the HC system forms the later part of this section. The section details how the model for the system is constructed. The HC system model, which is reflective of the actual HC system, serves as the external environment for the automata model.

4.1. Variable Structure Learning Automata

The variable structure learning automata is represented by the quintuple $\{\phi, \alpha, \beta, A, G\}$, where

1. The state of the automaton at any iteration n, denoted by $\phi(n)$ is an element of the finite set
$\phi = \{\phi_1, \phi_2, ..., \phi_s\}$
2. The output of the automaton at the iteration n, denoted by $\alpha(n)$, is an element of the set
$\alpha = \{\alpha_1, \alpha_2, ..., \alpha_r\}$.
3. The input to the automaton at the iteration n, denoted by $\beta(n)$, is an element of the set
$\beta = \{\beta_1, \beta_2, ..., \beta_m\}$.
4. A, is the updating algorithm or the reinforcement scheme.
5. The output function $G(.)$, determines the output of the automaton at any iteration n in terms of the state at that iteration.
$\alpha(n) = G[\phi(n)]$.

If each state of the quintuple corresponds to a unique action and the number of states are finite (as will be the case with our HC system), then $r = s < \infty$ and hence G is an identity mapping. Therefore, the automata can be represented as the triple $\{\alpha, \beta, A\}$.

An action probability $p_j(n)$ is associated with every action or state of the automaton at any iteration n. $p_j(n)$ represents the probability that the automata will be in state j at an iteration n, amongst the r possible states of the automaton. The updating algorithm or the reinforcement scheme, A, attempts to update the action probabilities at every iteration n and brings the automaton to an optimal solution state. The precise manner in which the action α_i performed at an iteration n and the response $\beta(n)$ of the environment change the probability $p_j(n)$, completely defines the reinforcement scheme. If $p_j(n+1)$ is a linear function of $p_j(n)$, then the reinforcement scheme is said to be linear, otherwise it is termed non-linear.

Reinforcement Scheme

Consider a variable-structure automaton with r actions to be operating in a stationary environment with $\beta = \{0,1\}$. In other words, the automaton is operating in a P-model environment, where the response is binary valued. Let N be the set of nonnegative integers and let $n \in N$. A general scheme to update the action probabilities can be represented as:

If $\alpha(n) = \alpha_i \quad (i = 1,2,...,r)$

$p_j(n+1) = p_j(n) - g_j[p(n)]$ when $\beta(n) = 0$

$p_j(n+1) = p_j(n) + h_j[p(n)]$ when $\beta(n) = 1$

for all $j \neq i$.

Now, we have $\sum_{j=1}^{r} p_j(n) = 1$, in order to preserve the probability measure. Therefore

$p_i(n+1) = p_i(n) + \sum_{j=1, j\neq i}^{r} g_j(p(n))$ when $\beta(n) = 0$.

$p_i(n+1) = p_i(n) - \sum_{j=1, j\neq i}^{r} h_j(p(n))$ when $\beta(n) = 1$.

Consider the variable structure automaton operating in the S-model environment, where the responses can take continuous values over an interval. By normalization, it can be assumed that the interval is a unit interval [0,1]. Hence, β for this model is a continuous variable where $\beta \in [0,1]$. The equations for the S-model can be written as:

$$p_i(n+1) = p_i(n) - (1-\beta(n)) * g_i(p(n)) + \beta(n) * h_i(p(n)) \quad \text{if } \alpha(n) \neq \alpha_i$$

$$p_i(n+1) = p_i(n) + (1-\beta(n)) * \sum_{j\neq i} g_j(p(n)) - \beta(n) * \sum_{j\neq i} h_j(p(n)) \quad \text{if } \alpha(n) = \alpha_i$$

The following assumptions are made with regard to the function g_j and $h_j (j = 1,2...,r)$.

Assumption 1 : g_j and h_j are continuous functions.

Assumption 2 : g_j and h_j are non-negative functions.

Assumption 3 : $0 < g_j(p) < p_j$

$0 < \sum_{j=1, j\neq i}^{r} [p_j + h_j(p)] < 1$

for all $i = 1,2,...r$ and all p whose elements are all in the open interval (0,1).

Different reinforcement schemes are represented by how the functions $g(.)$ and $h(.)$ are characterized. For instance if the functions are linear representations of the action probabilities, then scheme is linear, otherwise they are non-linear. It's also possible to have hybrid scheme's. Essentially, the nature of the reinforcement scheme affects the behavior of the model.

4.2. Construction of the Model

Figure 2 shows the schematic of the learning automata model. The model is constructed by associating every task s_i in the TFG with a variable structure automaton. Each of the automata are represented as a 3-tuple, $(\alpha^{s_i}, \beta^{s_i}, A^{s_i})$. Since the tasks can be assigned to any of the $|M|$ machines, the action set of the automata are identical. It is assumed that the environment is a S-model environment and hence the domain of the input set to the automata is in the interval $[0,1]$.

Therefore for any task s_i, $0 \leq i < |S|$:

$\alpha^{s_i} = m_1, m_2, ...m_{|M|-1}$

$\beta^{s_i} \in [0,1]$

where β^{s_i} *closer* to 0, indicates that the action taken by the automaton of task s_i was favorable to the system, and *closer* to 1 indicates an unfavorable response.

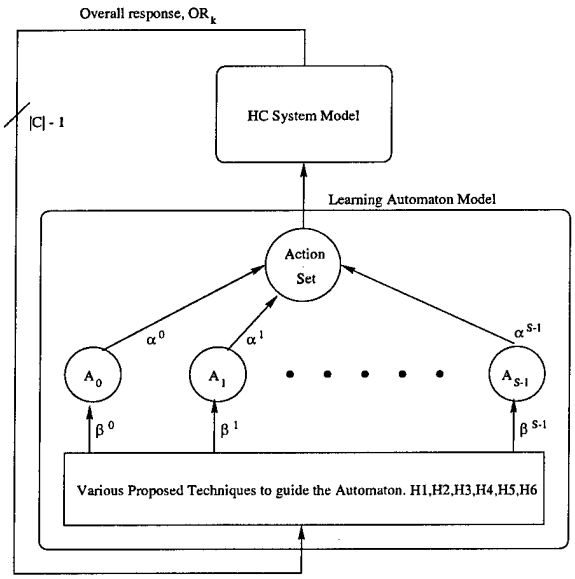


Figure 2. Learning Automata Model

The output response of the HC system model corresponding to each of the cost metrics in the set C, is assumed to be binary. In other words, if the value of a particular cost metric $c_k \in C$ at iteration n, is more optimal than its value at iteration $n-1$, then the output is favorable and if not the response is unfavorable.

If $c_k(n)$ is more optimal then $c_k(n-1)$

then $OR_k = 0$, else $OR_k = 1$ where $1 \leq k \leq |C|$

where OR_k is the overall output response of the system with

respect to $c_k(n)$.

The next step in developing the model is to translate OR_k into the inputs $\beta^{s_i}(n)$ for each of the automata. This involves two steps. First, OR_k would have to be translated into the input to each of the s_i automata. Let $\mu_k^{s_i}$ represent this value. Hence, $\mu_k^{s_i}(n)$ corresponds to the input to the automaton $(\alpha^{s_i}, \beta^{s_i}, A^{s_i})$, with respect to cost metric c_k at iteration n. Assume that λ_k indicates the weightage associated with the cost metric $c_k(n)$. Now the second step of this two step process can be represented as:

$\beta^{s_i}(n) = \sum_{k=1}^{|C|} \lambda_k * \mu_k^{s_i}$ for $i = 0$ *to* $|S| - 1$

where

$\sum_{k=1}^{|C|} \lambda_k = 1$ and $\lambda_k \geq 0$ for $k = 1$ *to* $|C|$

There are different ways of accomplishing the first step. The simplest and easiest way would be to make OR_k as the input to each of the automata in the model.

$H1 : \mu_k^{s_i}(n) = OR_k$ for all $s_i \in S$, and $c_k \in C$

$H1$ is simple to implement and completely ignores the structure of the TFG, while guiding the learning of the automata in the model. To put it differently, using $H1$ it is possible that an unfavorable output response from the system may be falsely interpreted as an unfavorable input to a particular automaton.

Ideally, the model should generate an assignment that results in a consistent favorable output response from the system. It is difficult to achieve this objective based on the structure of the TFG. To overcome this problem, the authors propose techniques based on the history of the actions generated. With every action of an automaton, two additional fields are associated. The first one indicates the number of favorable responses, and the second field indicates the number of unfavorable responses, corresponding to a particular action. Consequently:

$FAV \rightarrow$ Matrix of order $|S|$ x $|M|$ x $|C|$, the elements $fav[i, j, k]$ of which indicate the number of times a favorable response resulted when task s_i was assigned to machine m_j for cost metric c_k.

$UNFAV \rightarrow$ Matrix of order $|S|$ x $|M|$ x $|C|$, the elements $unfav[i, j, k]$ of which indicate the number of times an unfavorable response resulted when task s_i was assigned to machine m_j for cost metric c_k.

Based on this history information, different heuristic techniques can be developed that will help guide the automata to the optimal assignment. Five general heuristic techniques that the authors experimented with are presented here. Knowledge about the nature of an application can be exploited to develop heuritics specific to that application. But in this work, no assumptions are made with respect to the applications and consequently the heuristics can be used in any environment. The results of these techniques are compared with each other, and with the technique H1 that is not dependent on the structure of the TFG.

$H2$: If $fav[i, x_i(n), k] > unfav[i, x_i(n), k]$
then $\mu_k^{s_i}(n) = 0$, else $\mu_k^{s_i}(n) = 1$
for all $s_i \in S$, and $c_k \in C$

Since the objective is to maximize the number of favorable responses and minimize the unfavorable ones, this heuristic assigns an input to the automaton as favorable if for that particular action the number of favorable responses is greater than the unfavorable ones.

$H3$: If $(fav[i, x_i(n), k] - unfav[i, x_i(n)], k)$ is the maximum for all actions of $s_i \in S$, and $c_k \in C$
then $\mu_k^{s_i}(n) = 0$, else $\mu_k^{s_i}(n) = 1$

Here the difference in favorable and unfavorable responses is compared against all other actions. If the chosen action gives the maximum value, the automaton input is termed favorable.

$H4$:If($fav[i, x_i(n), k]$ - $min_{0 \leq p < |S|, 0 \leq q < |M|}$ $fav[p, q, k]$ $+max_{0 \leq p < |S|, 0 \leq q < |M|} unfav[p, q, k]$- $unfav[i, x_i(n), k]$) > ($max_{0 \leq p < |S|, 0 \leq q < |M|}$ $fav[p, q, k]$ - $fav[i, x_i(n), k]$ + $unfav[i, x_i(n), k]$ - $min_{0 \leq p < |S|, 0 \leq q < |M|}$ $unfav[p, q, k]$)
then $\mu_k^{s_i}(n) = 0$, else $\mu_k^{s_i}(n) = 1$,
for all $s_i \in S$, and $c_k \in C$

In this heuristic, if the chosen action has a high value for favorable responses and a low value for unfavorable responses, the input is favorable.

$H5$: If $fav[i, x_i(n), k] = max_{0 \leq p < |S|, 0 \leq q < |M|}$ $fav[p, q, k]$,
then $\mu_k^{s_i}(n) = 0$, else $\mu_k^{s_i}(n) = 1$,
for all $s_i \in S$, and $c_k \in C$

In H5, if the chosen action has the maximum favorable responses, the automaton input is favorable else it is unfavorable.

$H6$: If $unfav[i, x_i(n), k] = min_{0 \leq p < |S|, 0 \leq q < |M|}$ $unfav[p, q, k]$,
then $\mu_k^{s_i}(n) = 0$, else $\mu_k^{s_i}(n) = 1$,
for all $s_i \in S$, and $c_k \in C$

Similarily, in H6 if the chosen action has the least unfavorable responses the input is favorable.

In order to complete the model, action probabilities need to assigned for the actions of the automata. The action set α^{s_i}, of an automaton is equal to the set $|M|$ as discussed earlier. Hence, the action probability would be the probability of assigning a task s_i to one of the machines. Let this probility be $p_{ij}(n)$. Therefore:

$p_{ij}(n) \rightarrow$ the probability of assigning task s_i to machine m_j.

To preserve the probability measure, the sum of all the action probabilities for a particular task should equal one. Therefore:

$\sum_{j=0}^{|M|-1} p_{ij}(n) = 1$ for any $s_i \in S$

Reinforcement Scheme

The general reinforcement scheme for this model can now be formulated in terms of the action probabilities. Following the construction of the scheme from [1], consider the automaton for a task s_i:

$$p_{ij}(n+1) = p_{ij}(n) - ((1 - \beta^{s_i}(n)) * g(p_{ij}(n))) + (\beta^{s_i}(n) * h(p_{ij}(n)))$$
$$\text{for all } j \neq x_i(n)$$
$$p_{ix_i(n)}(n+1) = p_{ix_i(n)}(n) + ((1 - \beta^{s_i}(n)) * \sum_{j=0, j\neq x_i(n)}^{|M|-1} g(p_{ij}(n))) - (\beta^{s_i}(n) * \sum_{j=0, j\neq x_i(n)}^{|M|-1} h(p_{ij}(n)))$$

The nature of the functions $g(.)$ and $h(.)$ determine the nature of the reinforcement scheme and hence the performance of the scheduling system. These functions could be linear, non-linear or hybrid. For our study here, it is assumed that the functions are linear and are defined as:

$g(p_{ij}(n)) = a * (p_{ij}(n))$

and

$h(p_{ij}(n)) = b \,/\, (|M| - 1) - (b * p_{ij}(n))$

Parameters a and b are known as reward and penalty parameters respectively, and help guide the automaton to the optimal solution. The choice of these parameters affects the behavior of each of the proposed techniques. The next section details the performance analysis of these techniques.

5. Results and Discussion

In this section, the different heuristics that are proposed are analysed with respect to specific cost metrics. It begins with an introduction to the simulation environment and subsequently the results are presented and analysed.

The performance of the different techniques was evaluated by using randomly generated task graphs and processor graphs. The execution cost, communication time, data exchange and the power cost matrices were randomly generated over some predefined ranges with uniform probability. A specific set of cost metrics was then chosen and the performance evaluated for varying values of the reward and penalty parameters, a and b. The automata model was iterated until the probability of a chosen action in each automaton exceeded 0.99, or the number of iterations reached a maximum limit. If the former condition stopped the automata, then the model was said to converge. The latter condition meant the model was non-convergent. Table 1 shows the predefined ranges used to generate the random graphs. The efficiency of an algorithm is determined by the quality of the solution generated and the fastness of the algorithm.

Number of tasks	10
Number of machines	7
Number of edges	5,10,15,20 and 25
Execution matrix data range	1000
Communication matrix data range	4
Data exchange matrix data range	500
Power matrix data range	25

Table 1. Parameters for TFG and PG

Therefore, in order to study the performance of our model, the number of iterations required for convergence and the solution cost generated are studied as a function of the communication complexity.

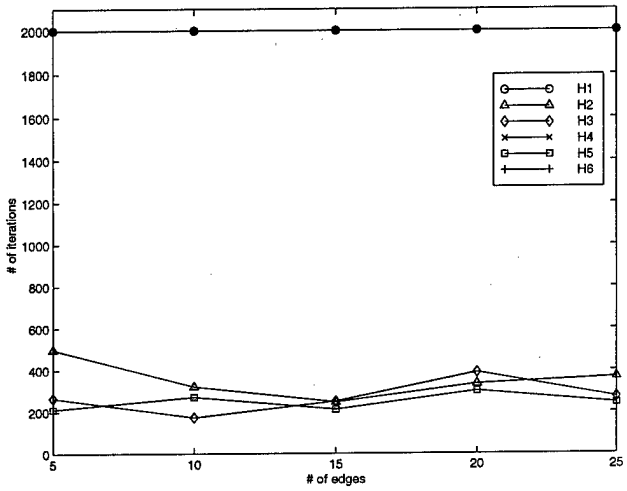


Figure 3. # of iterations vs Edges: a = b = 0.1

Initially, to verify the working of the model, a single cost metric c_1 (refer section 3) was chosen. Two sets of studies were performed. For the first one, the reward and penalty parameters a and b were made equal, $a = b = 0.1$, and the maximum limit for the number of iterations was set at 2000. For the second set, $a = 0.1$ and $b = 0.009$, and the limit for the number of iterations was set at 10000. Figures 3 and 4, show the results for the first set of data. From the graphs, it can be concluded that when $a = b$, techniques $H1$, $H4$ and $H6$ are non-convergent and heuristic $H2$ provides the best results for all communication complexities. Figures 5 and 6, show the results for the second set of experiments. Here, techniques $H4$ and $H6$ continue to remain non-convergent but $H1$ converges though it requires considerably higher number of iterations. The cost graph indicates $H1$ generates better results than $H2$.

The first set of experimental results validate the working of the model. The best solutions generated by the proposed techniques were close to the optimal solution. The results

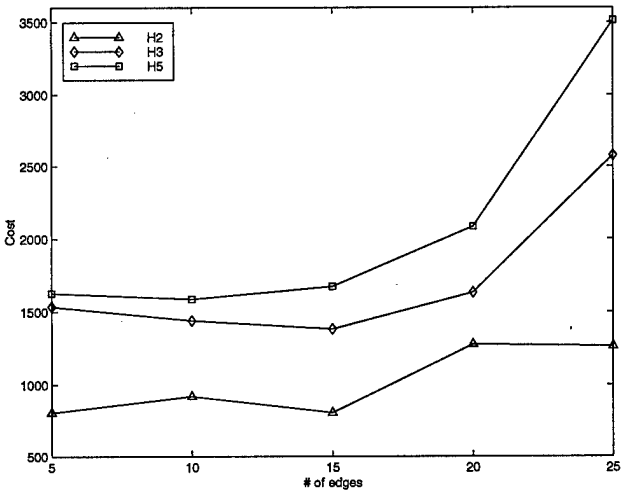


Figure 4. Cost vs Edges: a = b = 0.1

indicate that the choice of the reward and penalty parameters affect the solution cost and the choice of the heuristics.

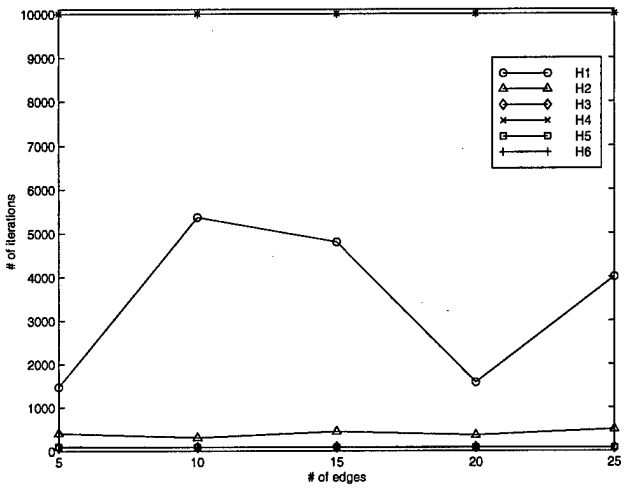


Figure 5. # of iterations vs Edges: a = 0.1,b = 0.009

In order to study the effectiveness of the model for multiple cost metric optimization, two cost metrics were chosen, c_1 and c_3(refer section 3). Experiments were performed by varying the weights associated with the metrics, λ_{c_1} and λ_{c_3} such that $\lambda_{c_1} + \lambda_{c_3} = 1$. The maximum limit for the number of iterations was set at 2000. The values for the reward and penalty parameters were chosen to get the best results. The results that were obtained have been tabulated in Tables 2,3,4 and 5. Techniques H4 and H6 were once again non-convergent. The effect of the weights associated with cost metrics can clearly be seen in all the tables. The first observation that can be made from the tables is that, as the weight associated with a particular cost metric is decreased the optimality of its solution also decreased. Essentially therefore, the automata model achieves independent optimization of the cost metrics subject to its weight. This is different from other works in that, the weighted sum of the metrics are optimized instead of independent optimization. With regards to the heuristic techniques, it can be seen that technique H1 produces the best results. For each of the graphs that were generated the optimal cost of metric c_1 was 985 and that of c_2 was 17. From the table it can be seen clearly that solution generated by H1 is close to the optimal. Technique H2 produced the next best results. Both H1 and H2 converge in almost the same number of iterations. This is in sharp contrast to techniques H3 and H5 which converge very fast. But the solution space generated by these techniques are far from the optimal. This is very pronounced as the communication complexity increases. But H1 on the other hand continued to produce optimal solutions even with increased communication complexity.

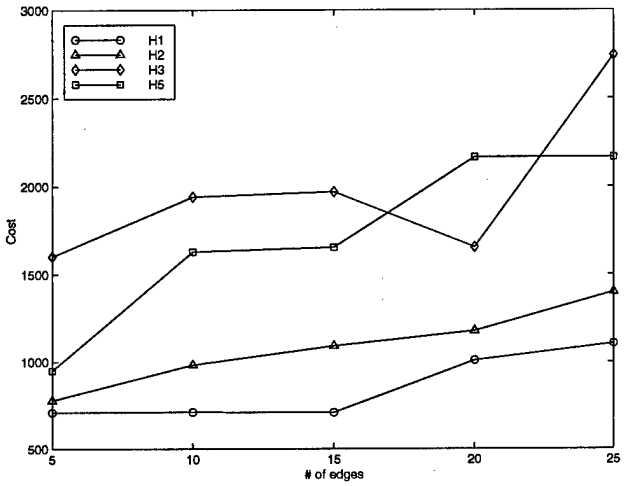


Figure 6. Cost vs Edges: a = 0.1,b = 0.009

6. Conclusions and Future Work

In conclusion, this work presented a framework for task assignment in heterogeneous computing systems. The framework was based on a learning automaton model and optimizes multiple cost metrics. The proposed model can be used for dynamic task assignment and scheduling. The model can also adapt itself to changing hardware environments. The cost metrics could be application specific or

# of edges	λ_{c_1}	λ_{c_3}	c_1	c_3	Iterations
5	1.00	0.00	985	51	420
	0.75	0.25	985	30	529
	0.50	0.50	1039	17	504
	0.25	0.75	1125	17	476
	0.00	1.00	1168	17	366
10	1.00	0.00	985	48	430
	0.75	0.25	985	34	460
	0.50	0.50	1103	17	549
	0.25	0.75	1192	17	383
	0.00	1.00	1265	17	342
15	1.00	0.00	985	53	453
	0.75	0.25	985	24	960
	0.50	0.50	1154	18	564
	0.25	0.75	1342	17	516
	0.00	1.00	1431	17	266
20	1.00	0.00	985	62	327
	0.75	0.25	985	27	1021
	0.50	0.50	1561	19	476
	0.25	0.75	1776	17	572
	0.00	1.00	1776	17	246
25	1.00	0.00	985	55	907
	0.75	0.25	1119	38	1390
	0.50	0.50	1460	20	460
	0.25	0.75	1911	17	456
	0.00	1.00	1776	17	246

Table 2. Cost values for Technique H1

# of edges	λ_{c_1}	λ_{c_3}	c_1	c_3	Iterations
5	1.00	0.00	1185	57	255
	0.75	0.25	1168	33	295
	0.50	0.50	1263	23	427
	0.25	0.75	1165	23	388
	0.00	1.00	1201	19	457
10	1.00	0.00	1156	57	317
	0.75	0.25	1168	37	341
	0.50	0.50	1192	29	266
	0.25	0.75	1265	19	340
	0.00	1.00	1215	19	457
15	1.00	0.00	1137	54	304
	0.75	0.25	1357	39	292
	0.50	0.50	1392	19	373
	0.25	0.75	1342	28	241
	0.00	1.00	1489	19	369
20	1.00	0.00	1692	59	306
	0.75	0.25	1634	58	389
	0.50	0.50	1561	30	336
	0.25	0.75	2061	19	226
	0.00	1.00	1909	19	443
25	1.00	0.00	1647	67	304
	0.75	0.25	1685	58	354
	0.50	0.50	1718	24	280
	0.25	0.75	1776	17	460
	0.00	1.00	1909	19	443

Table 3. Cost values for Technique H2

a general metric. The model was constructed by associating every task in the TFG with a learning automaton. The various automata work in unison to produce the optimal assignment of tasks. The paper presented various techniques to guide the learning of the automata. These techniques are not specific to any application and are applicable to any environment. The performance analysis of these techniques revealed that technique $H1$ results in the most optimal solution when both a single cost metric and multiple cost metrics are considered. It can therefore be concluded that for the proposed framework, the structure of the TFG does not adversely affect the task assignment procedure. However, techniques specific to a class of applications can be developed that will produce better results. Techniques H3 and H5 provide fast results but they are sub-optimal. As future work, it would be interesting to study how these techniques perform for application specific cost metrics. Also, a thorough analysis of the effect of the reward and penalty parameters in guiding the model to the optimal solution would be essential.

# of edges	λ_{c_1}	λ_{c_3}	c_1	c_3	Iterations
5	1.00	0.00	1541	61	81
	0.75	0.25	1119	79	85
	0.50	0.50	1257	68	86
	0.25	0.75	1489	34	72
	0.00	1.00	2446	32	81
10	1.00	0.00	1909	79	76
	0.75	0.25	1851	75	84
	0.50	0.50	1489	44	89
	0.25	0.75	2150	46	77
	0.00	1.00	2446	32	79
15	1.00	0.00	1793	95	80
	0.75	0.25	1975	105	80
	0.50	0.50	2075	59	75
	0.25	0.75	2470	44	79
	0.00	1.00	2472	32	78
20	1.00	0.00	3287	121	77
	0.75	0.25	2044	94	82
	0.50	0.50	2689	49	80
	0.25	0.75	3521	25	90
	0.00	1.00	2847	32	78
25	1.00	0.00	3287	121	77
	0.75	0.25	3001	184	81
	0.50	0.50	2689	42	76
	0.25	0.75	2061	46	86
	0.00	1.00	2847	32	79

Table 4. Cost values for Technique H3

References

[1] K. Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

# of edges	λ_{c_1}	λ_{c_3}	c_1	c_3	Iterations
5	1.00	0.00	1431	012	84
	0.75	0.25	1591	88	80
	0.50	0.50	1621	85	78
	0.25	0.75	1591	50	81
	0.00	1.00	1591	50	81
10	1.00	0.00	1621	97	75
	0.75	0.25	1591	88	85
	0.50	0.50	1633	85	78
	0.25	0.75	2280	50	81
	0.00	1.00	2280	50	81
15	1.00	0.00	1985	97	76
	0.75	0.25	1665	97	78
	0.50	0.50	2266	72	75
	0.25	0.75	2334	50	81
	0.00	1.00	2334	50	81
20	1.00	0.00	1993	92	80
	0.75	0.25	2949	83	79
	0.50	0.50	2266	85	79
	0.25	0.75	2255	56	76
	0.00	1.00	3948	50	81
25	1.00	0.00	2949	102	81
	0.75	0.25	2949	96	77
	0.50	0.50	2266	85	79
	0.25	0.75	2255	56	77
	0.00	1.00	3948	50	81

Table 5. Cost values for Technique H5

[2] M. M. Eshaghian. *Heterogeneous Computing,* Artech House, Norwood, MA, 1996.

[3] R. Mirchandaney and J. A. Stankovic. Using Stochastic Learning Automata for Job Scheduling in Distributed Processing Systems. *Journal of Parallel and Distributed Computing*, 3:527–552, 1986.

[4] B. Narahari et al. Matching and Scheduling in a Generalized Optimal Selection Theory. *Proceedings of Heterogeneous Computing Workshop*, 3–8, April 1994.

[5] H. S. Stone. Multiprocessor Scheduling with the aid of Network Flow Algorithms. *IEEE Trans. Software Eng.*, 3(1):85–93, January 1977.

[6] R. F. Freund. Optimal Selection Theory for Superconcurrency. *Proceedings Supercomputing '89*, 699–703, November 1989.

[7] R. F. Freund. SuperC or Distributed Heterogeneous HPC. *Computing Systems in Engineering*, 2(9):349–355, 1991.

[8] R. F. Freund and H. J. Seigel. Heterogeneous Processing. *IEEE Computer*, 26(6):13–17, June 1993.

[9] M. Wang et al. Augmenting the Optimal Selection Theory for Superconcurrency. *Proceedings of the Workshop on Heterogeneous Processing*, 13–22, 1992.

[10] S.Chen et al. Selection Theory for Methodology for Heterogeneous Supercomputing. *Proceedings of Heterogeneous Computing Workshop*, 5–22, April 1993.

[11] L. Wang et al. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing*, 47:8–22, November 1997.

[12] H. Singh and A. Youssef. Mapping and Scheduling Heterogeneous Task Graphs using Genetic Algorithms. *Proceedings of Heterogeneous Computing Workshop*, 86–97, April 1996.

[13] C. Hui and S. T. Chanson. Allocating Task Interaction Graphs to Processors in Heterogeneous Networks. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):908–923, September 1997.

[14] P. Shroff et al. Genetic Simulated Annealing for Scheduling Data-dependent tasks in Heterogeneous Environments. *Proceedings of Heterogeneous Computing Workshop*, 98–117, April 1996.

[15] H. J. Seigel et al. Heterogeneous Computing. *Parallel and Distributed Computing Handbook*, A.Y. Zomaya, McGraw-Hill, New York, 725–761, 1996.

[16] C. C. Shen and W. H. Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Trans. on Computer*, 34(3):197–203, March 1985.

Raju D. Venkataramana is currently pursuing a PhD in the Dept. of Computer Science and Engineering at the University of South Florida, Tampa. He received his B.E. in Computer Science and Engineering from Sri Venkateswara College of Engineering, University of Madras, India and Masters in Computer Science from the University of South Florida,Tampa in 1995 and 1997 respectively. His research interests include heterogeneous computing, parallel processing, interconnection networks and VLSI design.

N. Ranganathan is currently a Professor of Electrical and Computer Engineering at The Univ of Texas at El Paso, El Paso. He was previously on the faculty of the Center for Microelectronics Research and the Dept. of Computer Science and Engineering at the University of South Florida, Tampa (1988- Aug 1998), where he continues to hold a courtesy appointment as professor. His research interests include VLSI design and hardware algorithms, computer architecture and parallel processing.

On the robustness of metaprogram schedules

Ladislau Bölöni and Dan C. Marinescu
(Email: boloni, dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, IN, 47907, USA

Abstract

Scheduling meta applications on a computational grid uses estimation of the execution times of component programs to compute optimal schedules. In a realistic case various factors (hazards) lead to estimation errors, which affect both the performance of a schedule and resource utilization. We introduce the concept of robustness and present an analysis technique to determine the robustness of a schedule. We develop methods for reducing the chance that a metaprogram exceeds its execution time due to components outside its critical path. The results of this analysis are used to compute schedules less sensitive to hazards. This translates into more accurate reservation requirements for critical systems, and reduced expected execution time for non-critical metaprograms executed repeatedly. Simulation results prove the efficiency and applicability of our algorithms.

1. Introduction

Informally, a metaprogram is a collection of programs which cooperate towards a common goal. A computational grid is an abstraction for a collection of autonomous and heterogeneous computers interconnected by a high speed network. A metaprogram is executed on a computational grid. Throughout this paper we assume that there is no one scheduler which controls all resources of the system and that each local scheduler accepts reservations. We also assume that a meta scheduling agent [3] has information about the execution time of each component of the metaprogram and is capable to compute schedules. A schedule associates a node of the computing grid and a start up time to each component of the metaprogram. For an overview of high performance schedulers for a grid of autonomous computers and a comprehensive bibliography on the subject we refer the reader to [1].

There are two major challenges in metaprogram scheduling:

(a) The scheduling problem is NP-complete, therefore finding an optimal solution may be impossible or impractical except for trivial metaprograms and small grids. To overcome the explosion of the search space for realistic problems, one can apply approximation algorithms (heuristic-guided search), or genetic algorithms [5].

(b) The nondeterministic nature of the program execution time renders even an optimal solution approximate, or infeasible depending upon the resource allocation model. Several solutions to accommodate the nondeterministic execution time of the components of a metaprogram are possible:

(b1) Grossly overestimate the execution time of each program and minimize the risk of exceeding the alloted use of each host at the expense of the utilization of the grid,

(b2) Use dynamic algorithms which compute schedules at execution time. Once the data flow allows a program to be scheduled, gather information about the state of the grid and compute a new schedule for the remaining components of the metaprogram,

(b3) Use static scheduling algorithms to compute schedules for various scenarios, and at run time adopt the schedule which best fits the current conditions, [5]. If the grid is shared by multiple metaprograms this may not be feasible,

(b4) Use static algorithms for finding schedules less vulnerable to hazards i.e. more robust.

In this article we explore the last alternative and observe that it can be used in conjecture with any other approach for accommodating the nondeterministic program execution times.

We now introduce a formalism for metaprogram scheduling and the notations used throughout this paper:

N_i, A, B, C – the components of a metaprogram

H_j – the hosts of the grid

S_i – the schedules

0-7695-0107-9/99 $10.00 © 1999 IEEE

$RunTime(N_i, H_j)$ – the execution time of the component N_i on H_j

P_i – a path of the metaprogram

$\mathcal{P}$ – the set of paths of the metaprogram

$\mathcal{P}_c$ – the subset of the potentially critical paths

$\mathcal{P}_n$ – the subset of non-critical paths

$\mathcal{N}$ – the set of all components of the metaprogram

$\mathcal{N}_c$ – the subset of the potentially critical components

$\mathcal{N}_n$ – the subset of non-critical components

t_i – the estimated execution time of component i in schedule S

t'_i – the actual execution time of component i in schedule S

t_{s_i} – starting time of component i in schedule S

t_{f_i} – the completion/end time of component i in schedule S

t_{max_i} – the upper bound of the execution time of component i

$t_{spare_i}(A \rightarrow B)$ – the spare time of the link from component A to B

σ_i – the slack of component N_i

$\bar{\sigma}_i$ – the adjusted slack of component N_i

Given a directed acyclic graph $V(N, E)$ the nodes $N = \{N_1, N_2, ...N_n\}$ of this graph are called *components* and the edges *data paths*. The acyclic graph is called a *metaprogram*. An *ordering* of the nodes is a permutation P of the nodes $\{N_{P_1}, N_{P_2}...N_{P_n}\}$ which preserves the order of the nodes in the graph, i.e. if there is an edge N_iN_j in the graph, then $P_i < P_j$.

Given a *grid* $G = \{H_1, H_2, ..., H_k\}$ consisting of k hosts, and a metaprogram $V = (N, E)$ a *mapping of the component* N_i to the grid is a function associating a unique host in G to the program, $Map(N_i) = H_j$. The *mapping of the metaprogram to the grid* is a set:

$$Map(V, G) = \{Map(N_i)\} \ \forall N_i \in N \qquad (1)$$

We associate with each pair (N_i, H_j) with $N_i \in N$ and $H_j \in G$ a scalar called the *running time of program* N_i *on host* H_j, $t_i = RunTime(N_i, H_j)$. The *running time of the metaprogram components on the grid* is the set: $RunTime(V, G) = \{RunTime(N_i, H_j)\} \ \forall N_i \in N, \ H_j \in G$.

Given a metaprogram V, a grid G, an ordering of the programs, P, and a mapping $Map(V, G)$ we associate to each program a scalar value called the *startup time* $t_{s_i} = Start(N_i)$. The *startup time of the metaprogram components* is the set $Start(V) = \{Start(N_i)\} \ \forall N_i \in N$. The *completion/end time* of a component is defined as $End(N_i) = t_{f_i} = t_{s_i} + t_i$ and the set $End(V) = \{End(N_i)\} \ \forall N_i \in N$ is called the *the completion time of the metaprogram components*. The *goal component* of a metaprogram is the component whose result is the output of the entire computation, it is the last executed component, and its completion time t_{f_g} coincides with the completion time of the metaprogram.

The following restriction applies: the execution of any two components N_i and N_j of a metaprogram mapped onto the same host, $Map(N_i) = Map(N_j)$, cannot overlap:

$$P_i < P_j \Rightarrow t_{f_i} + t_i \leq t_{s_i} \qquad (2)$$

Given a metaprogram $V(N, E)$, the grid G, and the set $RunTime(V, G)$ a *schedule* of the metaprogram on the grid is the triplet consisting of the ordering, mapping and starting times of all the components of the metaprogram, $(P(N), Map(V, G), Start(V))$. The *total running time of a schedule S on grid G* is $T(V, G, S) = \max_i(t_{f_i}) \ \forall N_i \in N$.

The *deterministic optimal scheduling problem* - given a metaprogram V and the set of the running times of its components on a grid, $RunTime(V, G)$ find the schedule which minimizes the total running time, $T(V, G)$. The deterministic nature of a schedule is due to the fact that the values in the set $RunTime(V, G)$ are deterministic.

The *nondeterministic metaprogram scheduling problem* is a variant of the deterministic metaprogram scheduling problem when we assume that the execution times in the set $T(V, G, *)$ are random variables with known distributions and we try to find the schedule which minimizes the the total running time. For static scheduling approach, we can only hope to minimize the *mean execution time* over a number of runs.

In practice, the distribution of the execution time of a program is difficult if not impossible to obtain. Analytical expressions can only be obtained for some special cases unlikely to be of interest. A sample distribution requires empirical knowledge about the program and the *execution history of the program*.

Our model does not account for data migration delays because we are primarily concerned with coarse-grain distributed computing on a high-speed, low-latency network. The analysis may be extended however to models where data migration has a significant impact on the execution times.

In the next section we introduce the concept of robustness and describe a technique to determine the tolerance of a schedule to the hazards. An $O(n^2)$ time algorithm is given.

2. The robustness of a schedule

An example illustrates the concept if robustness of a schedule. In Figure 1 we present a metaprogram together with the estimated execution times of its components. These estimates refer to a reference computer, and should be adjusted to take into account the performance of

the computer on which the execution takes place. The communication delays are ignored. Suppose that we want to schedule the program on a grid consisting of H1, H2 and H3. The reference computer is H1, while the speed of H2 is 50% and of H3 25% of the reference.

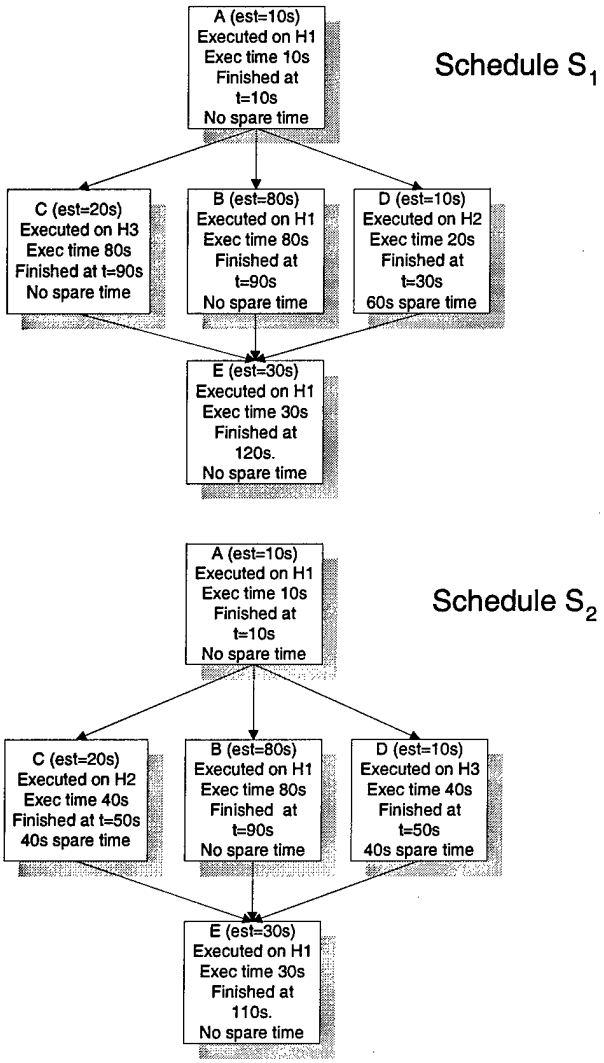


Figure 1. Two schedules of the same metaprogram with the same execution time but different robustness. The metaprograms are executed on the hosts H1, H2 and H3 with the relative speed 1.0, 0.5 and 0.25 respectively.

Consider two optimal schedules with the same execution time of $t = 120s$ for the given grid (we leave the proof of the optimality to the reader). A more attentive examination of the two schedules reveals an important difference among them. For S_1, component D finishes at the time $t = 30s$, while its output is needed at time $t = 90$ and the host on which it is scheduled (H2) is not used again in this metaprogram. In this case we say that the component D has 60 seconds of *spare time*. Even if the execution of D takes twice the estimated value, the total execution time of the metaprogram will not be affected. Unfortunately, none of the other components in this schedule have spare time; we say they are *critical*. For S_2 both C and D have 40 seconds of spare time.

Intuitively, it is obvious that the schedule on right, S_2, is "better" than S_1, the schedule on left of Figure 1. To provide a quantitative assessment of the difference between the two schedules assume a probability say $p = 0.2$ that a component is late and that the execution times are independent random variables. The last assumption may not be true in practice systems, however it is a good approximation in cases when the delays are caused by discrete events independent upon the distributed application.

The probability that the metaprogram is late for the two schedules are

$$p_{late}(S_1) = 1 - (1-p)^4 = 0.5904$$

$$p_{late}(S_2) = 1 - (1-p)^3 = 0.4880$$

If the metaprogram is executed repeatedly, then then the expected running time of a more robust schedule will be smaller than the expected running time of the less robust schedule, assuming that the running time of each component has small variations around its expected value.

Although our model of the execution times is naive, the qualitative result will apply to any reasonable narrow distribution of the execution times. We assumed a bounded execution time in order to prove the robustness of the algorithm. In practice a weaker assumption along the lines of a very narrow distribution around the estimated execution time should lead to the same result.

2.1. Data and host dependencies

In the following we devise an analytic measure of the vulnerability of a schedule to hazards. We are interested in the question how the increase in the execution time of a component affects the total execution time of the metaprogram, or in the positive effects of an early termination.

Given a component N_i of the metaprogram its starting, execution, and completion time are respectively t_{s_i}, t_i, and t_{f_i}. If the completion time is exceeded we say that component N_i is *late*. There are two reasons for a component to be late:

- The actual execution time of the component is longer than expected, $t'_i > t_i$, and/or
- The execution of the component begins later than expected: $t'_{s_i} > t_{s_i}$ due to interactions among the entities involved. We recognize *(a) data dependencies*

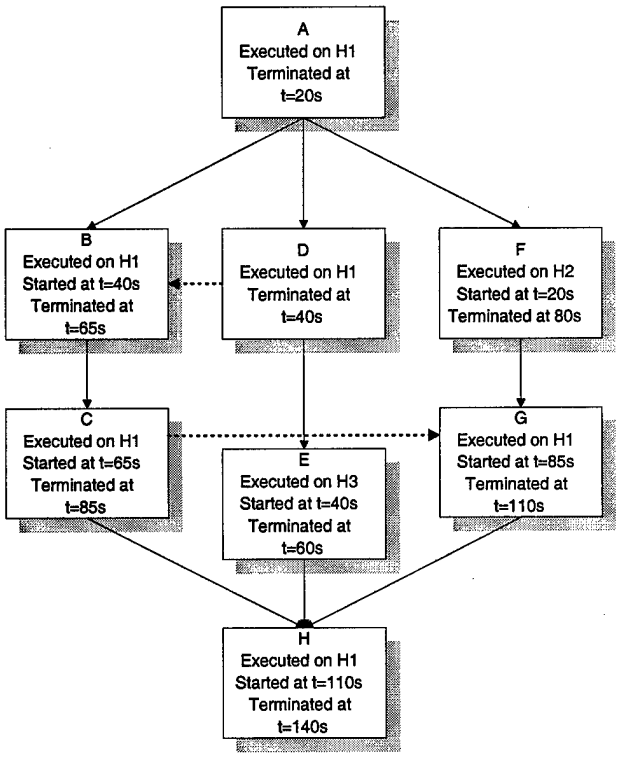


Figure 2. Augmentation of the data dependency graph of a metaprogram with links corresponding to the host dependencies. The host dependencies are drawn with dotted lines.

some component was late in creating data needed for the execution of component N_i, *(b) communication dependencies*, the data transfer between the producer and the consumer takes longer than expected, and *(c) host dependencies* some component was late in completing its execution on the same host where component N_i will be executed.

Host dependencies may be resolved by a dynamic scheduler, that can reassign the component to a different host. This is a nontrivial problem on its own, because the modifications in the mapping may lead to large performance penalties, and the real delay is not known at the moment of the scheduling. We may sacrifice our precomputed optimal schedule for an insignificant delay. In this paper we deal only with precomputed static schedules. While data and communication dependencies are invariants of the metaprogram, the host dependency is a property of a particular schedule.

In the following we introduce a robustness metrics for the schedules. In this case data and host dependencies can be treated identically. Our approach is to *augment the data dependency graph* of the metaprogram with the host dependency links as shown in Figure 2. If two programs are scheduled one after another on the same host, a new link is to be added between them. If a data dependency link between the two components is already in place no new link will be added. For optimal schedules most of the host dependencies follow the data dependencies because optimal schedules try to avoid moving data around.

Figure 2 illustrates the effect of host dependencies. Component C terminates at $t = 85s$, and the data it generates is needed only by component H which starts at $t = 110$, so one is tempted to believe that we have a comfortable spare time for C. However the host dependency link from C to G shows that any delay in terminating C will delay the start and implicitly the termination of component G on the host H1. Component G is critical, so the host dependency will make the component C critical, too.

In our analysis we do not differentiate between delays caused by a late start or longer execution time.

Given a metaprogram and a schedule, a *shifted schedule* is one where the mapping and order of execution of the components on every host is the same, but the startup of a component is adjusted such that it is launched immediately when its data and host dependencies are satisfied. We assume that the scheduler can automatically shift the schedule, if needed.

An upper limit of the effect of the delay is given by the following theorem.

Theorem 1 *Given a metaprogram V, a grid G, and a static schedule S, let the execution time of each component be t_i, and the total execution time be $T(V,G,S)$. If the actual execution time of each component changes by Δt_i there is a schedule S' whose execution time is smaller than:*

$$T(V,G,S) + \sum \Delta t_i'$$

where

$$\Delta t_i' = \begin{cases} \Delta t_i & \text{if } \Delta t_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

This theorem provides an upper limit for the delay of the schedules. However this upper limit is very disappointing, and raises the question if it cannot be improved. Consider a critical path $P_{crit} = \{C_1, C_2 ... C_n\}$ where C_1 is the starting component and C_n is the goal component of the metaprogram. Obviously a lower limit of the total execution time would be:

$$T(V,G,S) + \sum \Delta t_{C_i}$$

This lower limit shows us that if a component on the critical path is late, this delay will propagate into the total execution time of the metaprogram. A metaprogram can also

be late because of components outside the critical path. In this paper we develop methods for reducing the chance that a metaprogram is late due to components that are not on the critical path constructed assuming deterministic execution times.

Unfortunately, the influence of a late termination is more probable to affect the schedule than an early termination. This is due because most components have more than one dependency. Any late dependency forces the component to be late, while all dependencies should be early to permit an early start of the component.

2.2. Spare time

Consider a metaprogram and a static schedule. Given component A we want to determine the way in which the delay of component A influences the execution time of the metaprogram.

We assume that there is a path from each component to the goal component and there is only one goal component. If there are more then one, we introduce a *final goal component* which depends on all of the goal components, and has zero execution time.

The spare time of a link is defined as:

$$t_{spare}(A \rightarrow B) = t_{s_B} - t_{f_A}$$

The **slack of a component** σ_i is the minimum spare time on any path from the component to the goal component, or equivalently the length of the shortest path to the goal component in the augmented graph of spare times. We call a component **critical** if its slack is zero. Even if the spare time of a component is zero, its slack can be nonzero. The importance of the slack is demonstrated by the following theorem given here without a proof .

Theorem 2 *Consider a metaprogram M, a schedule S and a component N_i with a nonzero slack σ_i. If component N_i will exceed its estimated execution time by $\Delta t_i \leq \sigma_i$ the shifted schedule will have the same execution time as the original one, provided that all other components meet their deadlines.*

Corollary 1 *If a component N_i has at least one non-critical component on all paths to the goal state, the component is non-critical.*

This definition gives us a simple algorithm to compute the slack of all components of a metaprogram.

1. augment the metaprogram graph with the host dependency links.
2. label each link with the spare time on the link.
3. for each component compute the shortest path to the goal component in the graph of the spare times.

The spare time can be computed using Dijkstra's shortest path algorithm or improvements of it in $O(n^2)$ time [2]. The intuition behind the slack is that spare time on the shortest path from a component to the goal component may back-propagate and allow a component with no spare time to be late.

2.3. Adjusted slack

One of the drawbacks of using the slack of a measure of the robustness of a schedule, is the fact that every component is treated separately. Theorem 2 proves the slack of a component as an useful measure only for the case when all other components meet their deadlines, a rather strong assumption. In a practical case more than one component can be late. A delay in a component on which our component depends may cause a decrease in the slack of the current component. Intuitively, the same slack σ is better at the beginning of a computation, than close to the end, where possible delays from earlier components can "chop of" parts of it.

We are interested in a measure which proves a result similar to Theorem 2, but allows more than one component to be late. In the general case it is difficult to construct such a measure because it depends on the distribution of the execution time of the previous components. In the following we assume that the actual execution time of component N_i does not exceed its estimated execution time by more than a factor $q \geq 1, t'_i \leq q \times t_i$ and that q is the same for all components. This assumption bounds the starting time of a component as well

$$t'_{s_i} \leq q \times t_{s_i}$$

A bounded time metaprogram is one where the execution times of all components are bounded.

We define the *adjusted slack* as the slack modified to account for a late start.

$$\bar{\sigma}_i = \max(0, \sigma_i - (q-1)t_{s_i})$$

Theorem 3 *Consider a bounded time metaprogram V and a schedule S. If $\Delta t_i < \bar{\sigma}_i \ \forall N_i \in N$ then there is a shifted schedule S' with the same execution time as the original schedule S.*

This theorem shows that the adjusted slack is a more useful metric than the slack, because allows any of the components to be late within the limits of its adjusted slack.

We call a component *safe* if its adjusted slack is larger than the upper bound of delay on the component. A safe component can not cause the total execution time to be late (provided that the upper bound on delays holds). An immediate application of this analysis is the identification of the safe components.

2.4. Potentially critical paths and components

Call T_i the cost of a path P_i, h_i the cumulative effect of the hazards on that path, and $T_i' = T_i + h_i$ the cost of the path in the presence of the hazards, and write $P_i = (T_i, h_i)$.

The effect of the hazards partitions the set $\mathcal{P}$ of all paths into two disjoint subsets, $\mathcal{P}_c$ paths that have the potential of becoming critical path, and $\mathcal{P}_n$ paths that cannot become critical, $\mathcal{P} = \mathcal{P}_c \cup \mathcal{P}_n$ such that

$$\min(T_i', \forall P_i \in \mathcal{P}_c) > \max(T_j', \forall P_j \in \mathcal{P}_n)$$

We call a component N_i potentially critical if it appears on at least one potentially critical path. The set of components is $\mathcal{N} = \mathcal{N}_c \cup \mathcal{N}_n$ with $\mathcal{N}_c$ the set of potentially critical components and $\mathcal{N}_n$ components that are not critical.

The analysis supported by Theorems 2 and 3 can be restricted only to components in $\mathcal{N}_c$. Theorem 1 provides a tighter bound when applied only to components in $\mathcal{N}_c$. For example consider a metaprogram with 3 paths $P_1 = (1000, 10)$, $P_2 = (10, 600)$ and $P_3 = (60, 800)$. In this case $\mathcal{P}_c = \{P_1\}$ and $\mathcal{P}_n = \{P_2, P_3\}$. Assume that hazards effect only one component in each path, and these components are different. Applied to the entire set of components $\Delta t' = 1410$, and applied to $\mathcal{N}_c$ we have $\Delta t' = 10$.

The concept of robustness of a schedule can be expressed in terms of path criticality. If we call $p_i^{(h)}$ the probability that path P_i, $1 \leq i \leq n$ becomes critical subject to hazards h and $\sum_{i=1}^{n} p_i = 1$ then we can define the *entropy* of a schedule as

$$H^{(h)}(S) = -\sum_{i=1}^{n} p_i^{(h)} \log p_i^{(h)}$$

If a schedule with n paths has only one critical path say P_1 then $p_1 = 1, p_i = 0 \quad i = 2..n$ and we have $H(S) = 0$. If all n paths can become critical subject to hazards h then $p_i = \frac{1}{n}$ and $H(S) = \log n$.

The entropy of the path is then a measure of robustness. We say that schedule S_j is more robust than S_i if

$$H^{(h)}(S_i) < H^{(h)}(S_j)$$

Example:

Let us label the three paths of schedule S_1 in Figure 1 from left to right as P_1, P_2 and P_3. Assuming that the hazards lead to an increase of the execution time no more that 30s, we have

$$p_1 = p_2 = \frac{1}{2} \quad p_3 = 0 \quad H(S_1) = 1$$

For S_2 the paths are P_1', P_2' and P_3' and

$$p_1' = p_2' = 0 \quad p_3 = 1 \quad H(S_2) = 0$$

3 Constructing robust schedules

In this section we show how the robustness analysis can be used to improve a scheduling algorithm. Given a metaprogram V and a grid G we can partition the set of all schedules into equivalence classes based upon the total execution time of the metaprogram. Schedules with the same execution time form a class of iso-schedules, they either have the same critical path or equal cost (execution time) critical paths and in a deterministic case are indistinguishable from one one another but exhibit different performance under non-deterministic component execution time assumptions. In this paper we discuss metrics to differentiate amongst the members of a class based upon the robustness.

Two scenarios are studied: in the first one we assume a real time system where our goal is to maximize the number of safe components, while in the second one we consider a common metaprogram where the objective is to minimize the average execution time over a number of runs.

3.1 Scenario 1: Real time system

In this scenario we consider a real time distributed system, where meeting the deadlines is critical.

A component can be executed using static allocation and spatial partitioning of the grid. We call this case *strict scheduling conditions* and assume that a component scheduled this way will meet its deadline. An alternative way of executing a component on a grid is by dynamic allocation based upon temporal partitioning of the grid (*weak scheduling conditions*). This execution mode is more efficient from the point of view of resource utilization, but may cause delays in the execution of the components because a component may have to wait for a resource used by another component. We will assume that even under weak scheduling conditions we have an upper bound of the execution time $t_{max} = q \times t$.

Strict scheduling decreases the throughput of the system. We are interested in minimizing the number of components which need strict scheduling.

Theorem 3 shows that if the adjusted slack of a component is larger than the bound on execution time (safe components), the possible variation in the execution time can not affect the total execution time, regardless of the rest of

the system. This implies that a safe component does not require strict scheduling.

Our goal is to maximize the number of safe components, and to keep the estimated execution time smaller or equal than the deadline. An outline of a time-limited heuristic search robust algorithm is given below:

```
while(not timeout)
1. find a new schedule according to the
     heuristic
2. if the execution time is longer
     than the deadline, reject it
3. otherwise
  3a. perform robustness analysis
  3b. compute the num-
ber of safe components
  3c. if (number of safe compo-
nents larger
            than in the cur-
rent schedule)
        {
          make it the current schedule
        }
return current schedule
}
%
```

The same principle can be used to create a genetic algorithm for the same objective function.

Identification of safe components has important consequences upon resource utilization in case of strict scheduling conditions. Figure 3 shows the resource allocation graph for the components of the grid. The graph (a) shows the *static allocation* of all the resources of the grid for the time of execution of the metaprogram. The gray contour on the graph (b) shows the actual time intervals where different components are executed on the elements of the grid. The lower boundary of the contour is formed by the starting times of the first component, while the upper boundary by the finishing times of the last component executed on the specific host. Inside this we have a hashed contour which represent the time frame where critical components are executed. This region represents the time and space frame where resource allocation is required. In the gray region only safe components are executed, which does not require resource allocation, while in the white region no component of the metaprogram is executed. The hashed region corresponding to the strict resource allocation requirements still span the entire time between the start and end of the metaprogram - corresponding to the critical path of the schedule. The robustness analysis permits a *dynamic allocation of resources* by identifying components outside the critical path which do not require strict resource allocation and/or selecting the schedules which maximizes the number of such components.

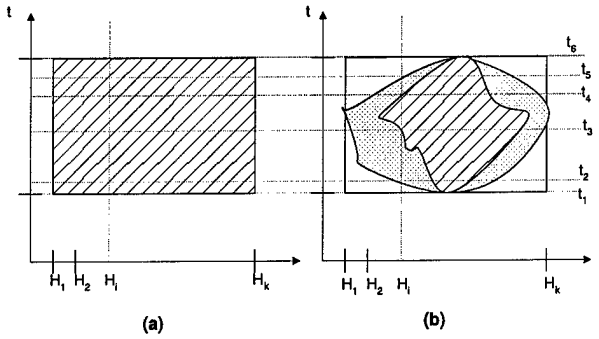


Figure 3. The effect of the robustness analysis upon the resource allocation. On the schedule on left host H_j is allocated exclusively to the metaprogram at time t_1 and deallocated at time t_6. On the schedule on the right the resource is available for other tasks during the intervals (t_1, t_2) and (t_5, t_6), it executes non-critical tasks in (t_2, t_3) and (t_4, t_5) and critical tasks in (t_3, t_4)

3.2 Scenario 2: Common metaprograms

In this scenario we are assuming that our system is not critical (i.e. there is no deadline). We are still assuming a bounded distribution of the execution time around the estimated value. In this case we are interested in obtaining a minimal average execution time for the distributed application for a number of runs.

To design a search algorithm we need a unique *robustness measure*, to compare schedules. A good robustness measure should have the following properties:

- Increase in the components slack.
- Penalize the slack larger than the time bound.

Unfortunately, devising a measure which accurately predict which of the iso-schedules gives the minimal average execution time depends on the shape of the distribution of execution times, an information difficult to obtain in practice. We are proposing an empirical formula which has the advantage of being simple, easy to compute, and performs well in experiments.

$$R(S) = \sum_i \frac{min(\sigma_i, t_i^{upper} - t_i)}{t_i^{\gamma}}$$

where $\gamma \in (0, 1]$ is a constant depending on the shape of the distribution and can be determined experimentally.

An outline of a timeout-limited heuristic search robust algorithm is given below:

```
while(not timeout)
  1. find a new schedule accord-
ing to the
       heuristic
  2. if the execu-
tion time is longer than
       the current schedule reject it
  3. if it is shorter
       make it the current schedule
  3. if they have equal length
       3.a perform robustness analysis
           on the new schedule
       3.b compute the robust-
ness measure
           R(S) for the new schedule
       3.c if it is larger than for the
           current schedule
           make it the cur-
rent schedule
return current schedule
```

4 Experimental results

In this section we study two questions:

- What is the cost of taking into consideration the robustness in scheduling?
- What is the average improvement achievable by robust scheduling?

The answer of these question depends on the nature of applications, the distribution of the execution times, so a general answer is difficult to give. An extensive testing process was performed using the Bond environment [6] to build up some confidence in the results.

All examples presented in these article were handcrafted in order to prove the validity of ideas. Nevertheless they did not answer an important question: how often in practice the possibility of improving the schedule using the robustness metric arises, and how dramatic these improvements are? These questions can not be answered without a knowledge of the application domain, some type of metaprograms may permit more optimizations than others.

Our experiments were made using randomly generated schedules. We were motivated by the desire to provide a fair evaluation of the algorithms provided in this paper. Carefully selected schedules may favor an algorithm with a poor performance on most schedules.

The random schedules were generated using following algorithm:

1. The number of components NC is generated as a random number in the range 15..30. The type of each component was chosen randomly from a collection of 5 different components.
2. The number of links are generated in the range 1.. NC*(NC-1)/2
3. The links are generated by generating random pairs of numbers. The connection always goes from lower ranking to higher ranking component, in order to avoid cycles.

In the testing process 100 random metaprograms were generated. For each metaprogram a time-out limited optimal static scheduling algorithm was applied. The algorithm was modified to maintain the optimal schedule according to the robustness measures presented for the two scenarios presented in Section 3. The robustness analysis was performed only if needed (i.e. if the decision could not be done using the total execution time). Every schedule was run 200 times with the execution times as a random normal distribution with the mean being the estimated execution time, and the upper bound being $q = 1.4$.

We have collected the following data:

- The number of schedules checked, and the number of iso-schedules found.
- The number of schedules for which the robustness analysis was performed.
- The best and worst number of safe components.
- The minimal average execution time, and the execution time of the first optimal schedule found (which would have been retained without the robustness analysis)

In our algorithm the robustness analysis was performed only when the total execution times were identical.

The robustness analysis was performed 129,683 times for the 8,768,794 schedules generated, representing less than 1.5% of the cases. We can conclude that the robustness analysis is easy to implement and cheap in respect to the computational demand.

The Table 4 shows the relative improvement in the number of safe components and mean execution time.

The results show an average of 67% increase in the number of safe components. The improvement in the execution

	without robustness analysis	with robustness analysis
No. of safe components	5.23	8.74
Mean execution time for 200 runs	68.93s	67.91s

Table 1. The effects of the robustness analysis on the number of safe components and the mean execution time of a metaprogram - the numbers represent the average of 100 randomly generated metaprograms

time is not as spectacular, being in the range of 1.5%. Nevertheless, 1.5% improvement in the execution time represents approximately 10% improvement in the expected delay. For the best case tested the number of safe components was improved from 3 to 16, and the mean execution time decreased by 6%. In the worst case there was no improvement in the indicators. The variation of the results is caused by the internal structure of the metaprograms. The robustness analysis cannot create spare times, it can only redistribute it to places where is needed, and even these redistribution has specific limitations.

As a side effect of the robustness analysis the safe components of a schedule are identified. Even if we can not increase their number, due to the specifics of the metaprogram, by identifying the safe components we can employ dynamic resource allocation, reducing the cost of running the metaprogram.

As a conclusion, robustness analysis is a practical method to improve the quality of metaprogram scheduling algorithms. We argue that the robustness analysis can be integrated into scheduling algorithms with ease and at a low cost and that the gain in terms of better quality schedules outweighs the computation and implementation costs.

5. Conclusions

Scheduling of dependent tasks with deterministic execution times is known to be NP complete. The problem of scheduling tasks whose execution time is non-deterministic as a result of various hazards is conceptually more challenging.

Given a schedule for an augmented dependency graph one can construct classes of iso-schedules, schedules with the same cost of the critical path (the cost of the critical path is the elapsed time from the initial component to the goal component). Some of the schedules in one class are less sensitive to the effects of the hazards because components on non-critical path have some spare time or slack in our terminology. The startup time of such a component can be delayed or its execution time may take longer in such a "shifted schedule", without increasing the total execution time of the entire graph. We call this property of a schedule robustness.

In this paper we assume bounded variations of the execution time of components and devise an analytic measure of the vulnerability of a schedule to hazards. We provide an upper bound for the execution time of a schedule subject to hazards and prove two theorems regarding shifted schedules.

We introduce the concept of a critical component, one whose increase of the execution time due to hazards may cause the execution path to become critical. Then we discuss measures of robustness. A possible measure is the number of critical components within a schedule, the fewer, the more robust the schedule. An alternative measure of robustness introduced in this paper is the entropy of a schedule. The entropy of a schedule is based upon the probability of an execution path of becoming critical. In the general case, determining this probability is a non-trivial task and more research is needed before this measure may prove its usefulness.

Elsewhere [4] we provide the proofs to the theorems in this paper.

6. Acknowledgments

The research reported here is partially supported by the National Science Foundation grant MCB-9527131, by the Scalable I/O Initiative, by a grant from the Computational Science Alliance, and by a grant from Intel Corporation.

References

[1] F. Berman. *High Performance Schedulers.* Morgan Kaufmann, 1998.

[2] D. Bertsekas. *Linear Network Optimization: Algorithms and Codes.* MIT Press, 1991.

[3] L. Bölöni, K. Jun, and D. Marinescu. Qos and reliability models for network computing. *Department of Computer Sciences, Purdue University CSD-TR #97-051*, 1997.

[4] L. Bölöni and D. Marinescu. Robust scheduling of metaprograms. *Purdue University CSD-TR #98-003, 1998 (submitted)*, 1998.

[5] J. Budenske, R. Ramanukan, and H. Siegel. On-line use of off-line derived mappings for iterative automatic target recognition tasks and a particular class of hardware platforms. *Proceedings of the HCW'97 Heterogeneous Computing Workshop*, pages 74–82, April 1997.

[6] D. Marinescu and L. Bölöni. Reflections on metacomputing, the bond view. *Purdue University CSD-TR #98-006*, 1998.

Ladislau L. Bölöni is a PhD student and Research Assistant in the Computer Sciences Department at Purdue Uni-

versity. He received a Diploma Engineer degree in Computer Engineering with Honors from the Technical University of Cluj-Napoca, Romania in 1993. He received a fellowship from the Hungarian Academy of Sciences for the 1994-95 academic year. He is a member of ACM and the Upsilon Pi Epsilon honorary society. His research interests include distributed object systems, autonomous agents and parallel computing. He can be contacted at boloni@cs.purdue.edu.

Dan C. Marinescu is Professor of Computer Science at Purdue University. He has been involved in scientific applications of high performance computing since early 80's when he was the principal architect of a real-time data acquisition and analysis systems used in the experiments leading to the discovery of super-heavy elements. Currently Prof. Marinescu is the Principal Investigator of an NSF funded Grand Challenge project to solve large structural biology problems using parallel and distributed computers and the Director of the Bond project (http://bond.cs.purdue.edu). He has co-authored more than 100 technical papers in distributed systems and networking, performance evaluation, parallel processing, scientific computing, and Petri Nets. He has an MS and PhD in electrical engineering from the Polytechnic Institute Bucharest, and an MS in EECS from U.C. Berkeley. Contact him at the Computer Sciences Department, Purdue University, West Lafayette, In, 47907, Email: dcm@cs.purdue.edu.

A Unified Resource Scheduling Framework for Heterogeneous Computing Environments

Ammar H. Alhusaini * and Viktor K. Prasanna*
Department of EE-Systems, EEB 200C
University of Southern California
Los Angeles, CA 90089-2562
Ph: (213) 740-4483
{ammar + prasanna}@usc.edu

C.S. Raghavendra
The Aerospace Corporation
P. O. Box 29257
Los Angeles, CA 90009
Ph: (310) 336-1686
raghu@aero.org

Abstract

A major challenge in Metacomputing Systems (Computational Grids) is to effectively use their shared resources, such as compute cycles, memory, communication network, and data repositories, to optimize desired global objectives. In this paper we develop a unified framework for resource scheduling in metacomputing systems where tasks with various requirements are submitted from participant sites. Our goal is to minimize the overall execution time of a collection of application tasks. In our model, each application task is represented by a Directed Acyclic Graph (DAG). A task consists of several subtasks and the resource requirements are specified at subtask level. Our framework is general and it accommodates emerging notions of Quality of Service (QoS) and advance resource reservations. In this paper, we present several scheduling algorithms which consider compute resources and data repositories that have advance reservations. As shown by our simulation results, it is advantageous to schedule the system resources in a unified manner rather than scheduling each type of resource separately. Our algorithms have at least 30% improvement over the separated approach with respect to completion time.

1. Introduction

With the improvements in communication capability among geographically distributed systems, it is attractive to use diverse set of resources to solve challenging applications. Such Heterogeneous Computing (HC) systems [12, 17] are called *metacomputing* systems [26] or *computational grids* [8]. Several research projects are underway, including for example, MSHN [22], Globus [13], and Legion [19], in which the users can select and employ resources at different domains in a seamless manner to execute their applications. In general, such metacomputing systems will have compute resources with different capabilities, display devices, and data repositories all interconnected by heterogeneous local and wide area networks. A variety of tools and services are being developed for users to submit and execute their applications on a metacomputing system.

A major challenge in using metacomputing systems is to effectively use the available resources. In a metacomputing environment, applications are submitted from various user sites and share system resources. These resources include compute resources, communication resources (network bandwidth), and data repositories (file servers). Programs executing in such an environment typically consist of one or more subtasks that communicate and cooperate to form a single application. Users submit jobs from their sites to a metacomputing system by sending their tasks along with Quality of Service (QoS) requirements.

Task scheduling in a distributed system is a classic problem (for a detailed classification see [5, 6]). Recently, there have been several works on scheduling tasks in metacomputing systems. Scheduling independent jobs (meta-tasks) has been considered in [2, 11, 14]. For application tasks represented by Directed Acyclic Graphs (DAGs), many dynamic scheduling algorithms have been devised. These include the Hybrid Remapper [20], the Generational algorithm [9], as well as others [15, 18]. Several static algorithms for scheduling DAGs in metacomputing systems are described in [16, 23, 24, 25, 27]. Most of the previous algorithms focus on compute cycles as the main resource. Also, previous DAGs scheduling algorithms assume that a subtask receives all its input data from its predecessor subtasks. Therefore, their scheduling decisions are based on machine performance for the subtasks and the cost of receiving input

*Supported by the DARPA/ITO Quorum Program through the Naval Postgraduate School under subcontract number N62271-97-M-0931.

0-7695-0107-9/99 $10.00 © 1999 IEEE

data from predecessor subtasks only.

Many metacomputing applications need other resources, such as data repositories, in addition to compute resources. For example, in data-intensive computing [21] applications access high-volume data from distributed data repositories such as databases and archival storage systems. Most of the execution time of these applications is in data movement. These applications can be computationally demanding and communication intensive as well [21]. To achieve high performance for such applications, the scheduling decisions must be based on all the required resources. Assigning a task to the machine that gives its best execution time may result in poor performance due to the cost of retrieving the required input data from data repositories. In [4], the impact of accessing data servers on scheduling decisions has been considered in the context of developing an AppLes agent for the Digital Sky Survey Analysis (DSSA) application. The DSSA AppLes selects where to run a statistical analysis according to the amount of required data from data servers. However, the primary motivation was to optimize the performance of a particular application.

In this paper we develop a unified framework for resource scheduling in metacomputing systems. Our framework considers compute resources as well as other resources such as the communication network and data repositories. Also, it incorporates the emerging concept of *advance reservations* where system resources can be reserved in advance for specific time intervals. In our framework, application tasks with various requirements are submitted from participant sites. An application task consists of subtasks and is represented by a DAG. The resource requirements are specified at the subtask level. A subtask's input data can be data items from its predecessors and/or data sets from data repositories. A subtask is ready for execution if all its predecessors have completed, and it has received all the input data needed for its execution. In our framework, we allow for input data sets to be replicated, i.e., the data set can be accessed from one or more data repositories. Additionally, a task can be submitted with QoS requirements, such as needed compute cycles, memory, communication bandwidth, maximum completion time, priority, etc. In our framework, sources of input data and the execution times of the subtasks on various machines along with their availability are considered simultaneously to minimize the overall completion time.

Although our unified framework allows many factors to be taken into account in resource scheduling, in this paper, to illustrate our ideas, we present several heuristic algorithms for a resource scheduling problem where the compute resources and the data repositories have advance reservations. These resources are available to schedule subtasks only during certain time intervals as they are reserved (by other users) at other times. QoS requirements such as deadlines and priorities will be included in future algorithms. The objective of our resource scheduling algorithms is to minimize the overall completion time of all the submitted tasks.

Our research is a part of the MSHN project [22], which is a collaborative effort between DoD (Naval Postgraduate School), academia (NPS, USC, Purdue University), and industry (NOEMIX). MSHN (Management System for Heterogeneous Networks) is designing and implementing a Resource Management System (RMS) for distributed heterogeneous and shared environments. MSHN assumes heterogeneity in resources, processes, and QoS requirements. Processes may have different priorities, deadlines, and compute characteristics. The goal is to schedule shared compute and network resources among individual applications so that their QoS requirements are satisfied. Our scheduling algorithms, or their derivatives, may be included in the Scheduling Advisor component of MSHN.

This paper is organized as follows. In the next section we introduce our unified resource scheduling framework. In Section 3, we present several heuristic algorithms for solving a general resource scheduling problem which considers input requirements from data repositories and advance reservations for system resources. Simulation results are presented in Section 4 to demonstrate the performance of our algorithms. Finally, Section 5 gives some future research directions.

2. The Scheduling Framework

2.1. Application Model

In the metacomputing system we are considering, n application tasks, $\{T_1, \ldots, T_n\}$, compete for computational as well as other resources (such as communication network and data repositories). Each application task consists of a set of communicating subtasks. The data dependencies among the subtasks are assumed to be known and are represented by a Directed Acyclic Graph (DAG), $G = (V, E)$. The set of subtasks of the application to be executed is represented by $V=\{v_1, v_2, \ldots, v_k\}$ where $v_k \geq 1$, and E represents the data dependencies and communication between subtasks. e_{ij} indicates communication from subtask v_i to v_j, and $|e_{ij}|$ represents the amount of data to be sent from v_i to v_j. Figure 1 shows an example with two application tasks. In this example, task 1 consists of three subtasks, and task 2 consists of nine subtasks.

In our framework, QoS requirements are specified for each task. These requirements include needed compute cycles, memory, communication bandwidth, maximum completion time, etc. In our model, a subtask's input data can be data items from its predecessors and/or data sets from data repositories. All of a subtask's input data (the data items and the data sets) must be retrieved before its execution. After

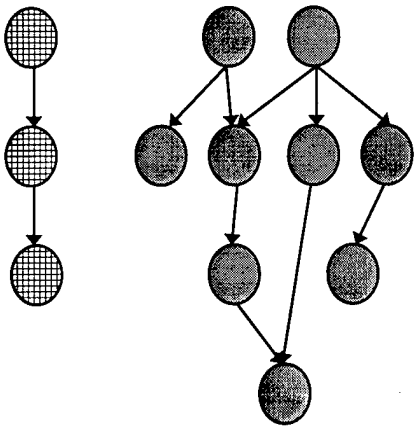


Figure 1. Example of application tasks

a subtask's completion, the generated output data may be forwarded to successor subtasks and/or written back to data repositories.

In some applications, a subtask may contain sub-subtasks. For example, Adaptive Signal Processing (ASP) applications are typically composed of a sequence of computation stages (subtasks). Each stage consists of a number of identical sub-subtasks (i.e., FFT's, QR decompositions, etc.). Each stage repeatedly receives its input from the previous stage, performs computations, and sends its output to the next stage.

2.2. System Model

The metacomputing system consists of m heterogeneous machines, $M = \{m_1, m_2, \ldots, m_m\}$, and f file servers or data repositories, $S = \{s_1, s_2, \ldots, s_f\}$. We assume that an estimate of the execution time of subtask v_i on machine m_j is available at compile-time. These estimated execution times are given in matrix ECT. Thus, $ECT(i, j)$ gives the estimated computation time for subtask i on machine j. If subtask v_i cannot be executed on machine m_j, then $ECT(i, j)$ is set to infinity.

System resources may not be available over some time intervals due to advance reservations. Available time intervals for machine m_j are given by $MA[j]$. Available time intervals for data repository s_j are given by $SA[j]$. Matrices TR and L give the message transfer time per byte and the communication latency between machines respectively. Matrices $Data\_TR$ and $Data\_L$ specify the message transfer time per byte and the communication latency between the data repositories and the machines, respectively. $DataSet[i]$ gives the amount of input data sets needed from data repositories for subtask v_i. In systems with multiple copies of data sets, one or more data repository can provide the required data sets for that subtask.

2.3. Problem Statement

Our goal is to minimize the overall execution time for a collection of applications that compete for system resources. This strategy (i.e., optimizing the performance of a collection of tasks as opposed to that of a single application) has been taken by SmartNet [11] and MSHN [22]. On the other hand, the emphasis in other projects such as AppLes [3] is to optimize the performance of an individual application rather than to cooperate with other applications sharing the resources. Since multiple users share the resources, optimizing the performance of an individual application may dramatically affect the completion time of other applications.

We now formally state our resource scheduling problem.

Given:

- A Metacomputing system with m machines and f data repositories,
- Advance reserved times for system resources as given by MA and SA,
- n application tasks, $\{T_1, \ldots, T_n\}$,where each application is represented by a DAG,
- Communication latencies and transfer rates among the various resources in matrices TR, L, $Data\_TR$, and $Data\_L$,
- Subtasks execution times on various machines in matrix ETC, and
- Amount of input data sets needed from data repositories for each subtask v_i as given by $DataSet[i]$.

Find a schedule to

$$Minimize\ \{\max_{j=1}^{n} [Finish\ Time(T_j)]\ \},$$

where the schedule determines, for each subtask, the start time and the duration of all the resources needed to execute that subtask.

Subject to the following constraints:

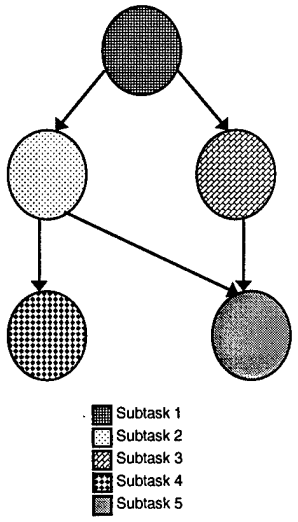


Figure 2. Application DAG for the example in Sec. 2.4

	m_1	m_2	m_3
V_1	5	4	8
V_2	20	5	3
V_3	6	10	4
V_4	10	4	2
V_5	∞	6	5

Table 1. Subtask execution times

- A subtask can execute only after all its predecessors have completed, all input data items have been received from its predecessors, and the input data sets have been retrieved from one of the data repositories,
- Preserve all advance resource reservations,
- Only one subtask can execute on any machine at any given time, and
- At most one subtask can access any data repository at any given time.

2.4. Separated Scheduling Vs. Unified Scheduling

Many scheduling methods exist in the literature for scheduling application DAGs on compute and network resources. They do not consider data repositories. With the inclusion of data repositories, one can obtain schedules for compute resources and data repositories independently and

	m_1	m_2	m_3
S_1	5	6	6
S_2	1	4	3
S_3	4	1.5	5

Table 2. Transfer costs (time units/data unit)

Subtask	Amount of the Input Data Set	Data Repository Choices
V_1	3 units	S_1 or S_2
V_2	10 units	S_2 or S_3
V_3	2 units	S_1 or S_3
V_4	1 unit	S_1 or S_2
V_5	5 units	S_3

Table 3. Input requirements for the subtasks

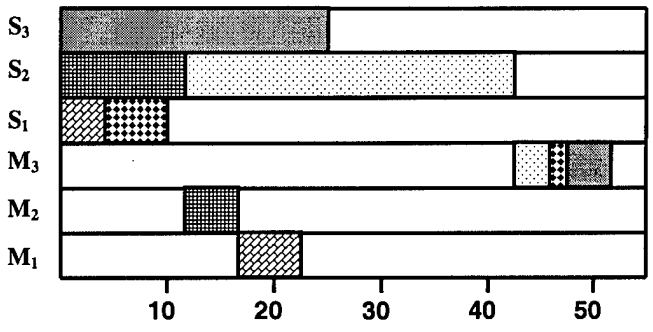


Figure 3. Separated scheduling (machines first)

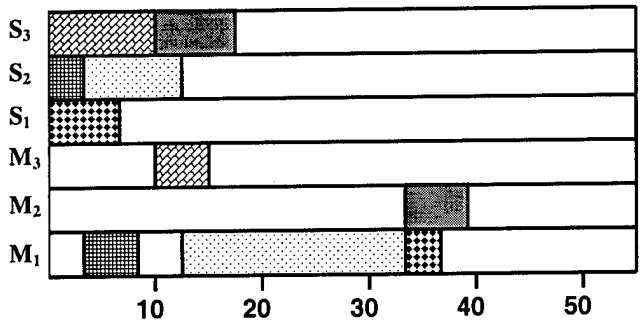


Figure 4. Separated scheduling (data repositories first)

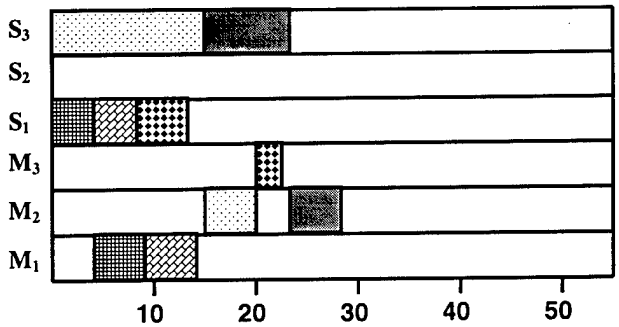


Figure 5. Unified scheduling

combine the schedules. In this section we show with a simple example, that this separated approach is not efficient with respect to completion time.

Figure 2 shows the DAG representation for an application task with 5 subtasks. In this example, we assume a fully connected system with 3 machines and 3 data repositories (file servers). The subtask execution times (in time units) are given in Table 1. Table 2 gives the the cost (in time units) for transferring one data unit from any data repository to any machine. We assume that each subtask needs an input data set, which can be retrieved from one or more data repositories as given in Table 3.

In this example, we are using a simple list scheduling algorithm called the Baseline Algorithm. This algorithm has been described in [20, 27]. The baseline algorithm is a fast static algorithm for mapping DAGs in HC environments. It partitions the subtasks in the DAG into blocks (levels) using an algorithm similar to the level partitioning algorithm which will be described in Section 3.1. Then all the subtasks are ordered such that the subtasks in block k come before the subtasks in block b, where $k < b$. The subtasks in the same block are sorted in descending order based on the number of descendents of each subtask (ties are broken arbitrarily). The subtasks are considered for mapping in this order. A subtask is mapped to the machine that gives the minimum completion time for that particular subtask. Since the original algorithm does not account for the data repositories, we implemented a modified version of the algorithm. In the modified version, the algorithm chooses a data repository that gives the best retrieving time of the input data set.

The schedule based on the separated approach, when scheduling the machines first, is shown in Figure 3. The completion time of this schedule is 52 time units. For this case, we map the application subtasks to the machines as they are the only resources in the system. Then for each subtask we choose the data repository that gives the best retrieving (delivery) time of the input data set to the previously mapped machine for this subtask in order to minimize its completion time. The completion time of the schedule based on the separated approach, when scheduling the data repositories first, is 39 time units as shown in Figure 4. For this case, we map the application subtasks to the data repositories as they are the only system resources. Then for each subtask we choose the machine that gives the best completion time for that subtask when using the previously mapped data repository to get the required data set for this subtask. Figure 5 shows the schedule based on the unified approach. The completion time of the unified scheduling is 28.5 time units. In the unified approach, we map each subtask to a machine and data repository at the same time in order to minimize its completion time.

The previous example shows clearly that the scheduling based on the separated approach is not efficient with respect to completion time. Further, with advance reservations, separated scheduling can lead to poor utilization of resources when one type of resource is not available while others are available.

3. Resource Scheduling Algorithms

In this section, we develop static (compile-time) heuristic algorithms for scheduling tasks in a metacomputing system where the compute resources and the data repositories have *advance reservations*. These resources are available to schedule subtasks only during certain time intervals as they are reserved (by other users) at other times. Although our framework incorporates the notion of QoS, the algorithms we present in this paper do not consider QoS. We are currently working on extending our scheduling algorithms to

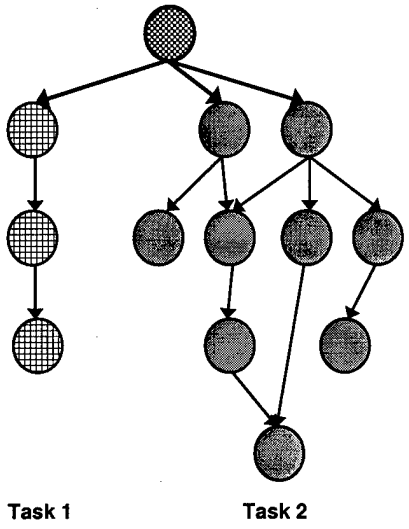


Figure 6. Combined DAG for the tasks in Fig. 1

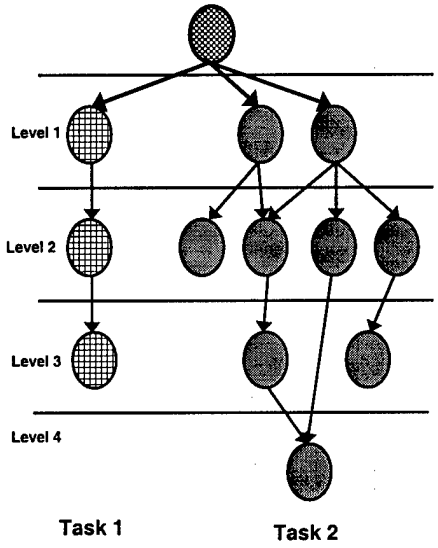


Figure 7. Level partitioning for the combined DAG in Fig. 6

consider QoS requirements such as deadlines, priorities, and security.

As in state-of-the-art systems, we assume a central scheduler with a given set of static application tasks to schedule. With static applications, the complete set of task to be scheduled is known a priori. Tasks from all sites are sent to the central scheduler to determine the schedule for each subtask so that the global objective is achieved. The information about the submitted tasks as well as status of various resources are communicated to the central scheduler. This centralized scheduler will then make appropriate decisions and can achieve better utilization of the resources.

Scheduling in metacomputing systems, even if we schedule based on compute resources only, is known to be NP-complete. One method is based on the well known list scheduling algorithm [1, 16, 23]. In list scheduling, all the subtasks of a DAG are placed in a list according to some priority assigned to each subtask. A subtask cannot be scheduled until all its predecessors have been scheduled. Ready subtasks are considered for scheduling in order of their priorities. In this section, we develop modified versions of list scheduling algorithm for our generalized task scheduling problem with advance resource reservations. Our heuristic algorithms that are based on the list scheduling are of two types – level by level scheduling and greedy approach. In the following, we briefly describe these two types of algorithms.

3.1. Level-By-Level Scheduling

In our framework, application tasks are represented by DAGs where a node is a subtask and the edges from predecessors represent control flow. Each subtask has computation cost, data items to be communicated from predecessor subtasks, and data sets from one or more repositories. A subtask is ready for execution if all its predecessors have completed, and it has received all the input data needed for its execution. To facilitate the discussion of our scheduling algorithms, a hypothetical node is created and linked, with zero communication time links, to the root nodes of all the submitted DAGs to obtain one combined DAG. This dummy node has zero computation time. Figure 6 shows the combined DAG for the two tasks in Figure 1. Now, minimizing the maximum time to complete this combined DAG achieves our global objective.

In level-by-level heuristic, we first partition the combined DAG into l levels of subtasks. Each level contains independent subtasks, i.e., there are no dependencies between the subtasks in the same level. Therefore, all the subtasks in a level can be executed in parallel once they are ready. Level 0 contains the dummy node. Level 1 contains all subtasks that do not have any incident edges originaly, i.e., subtasks without any predecessors in the original DAGs. All subtasks in level l have no successors. For each subtask v_j in level k, all of its predecessors are in levels 0 to $k-1$, and at least one of them in level k-1. Figure 7 shows the levels of the combined DAG in Fig. 6. The combined DAG in this example

```
Level-by-Level Scheduling Algorithm
begin
    Combine all submitted DAGs into one DAG.
    Do level partitioning for the combined DAG.
    For level := 1 to l do
        Set Ready to be the set of all subtasks at this level.
        While Ready is not empty do
            Find FINISH(v_i, m_min, s_min) for all subtasks in Ready, where m_min is
              the machine that gives the minimum completion time for subtask v_i
              if data repository s_min has been used to get the input data set.
            Min-FINISH: Choose the subtask v_k with the minimum completion time.
            Max-FINISH: Choose the subtask v_k with the maximum completion time.
            Schedule subtask v_k to machine m_min and data repository s_min.
            Update MA(m_min) and SA(s_min).
            Remove v_k from Ready.
        end While
    end For
end
```

Figure 8. Pseudo code for the level-by-level scheduling algorithms

has 4 levels.

The scheduler considers subtasks in each level at a time. Among the subtasks in a particular level i, the subtask with the minimum completion time will be scheduled first in the *Min-FINISH* algorithm and the subtask with the maximum completion time is scheduled first in the *Max-FINISH* algorithm. The advance reservations of compute resources and data repositories are handled by choosing the first-fit time interval to optimize the completion time of a subtask.

The idea behind the *Min-FINISH* algorithm, as in algorithm D in [14] and Min-min algorithm in SmartNet [11], is that at each step, we attempt to minimize the finish time of the last subtask in the ready set. On the other hand, the idea in the *Max-FINISH*, as in algorithm E in [14] and Max-min algorithm in SmartNet [11], is to minimize the worst case finishing time for critical subtasks by giving them the opportunity to be mapped to their best resources. The pseudo code for the level-by-level scheduling algorithms is shown in Figure 8.

3.2. Greedy Approach

Since the subtasks in a specific level i of the combined DAG belong to different independent tasks, by scheduling level by level we are creating dependency among various tasks. Further, the completion times of levels of different tasks can vary widely, and the level-by-level scheduling algorithms may not perform well. The idea in the greedy heuristics, *Min-FINISH-ALL* and *Max-FINISH-ALL*, is to consider subtasks in all the levels that are ready to execute in determining their schedule. This will advance execution of different tasks by different amounts and will attempt to achieve the global objective and provide good response times for short tasks at the same time. As before, we consider both the minimum finish time and the maximum finish time of all ready subtasks in determining the order of the subtasks to schedule.

The two greedy algorithms, *Min-FINISH-ALL* and *Max-FINISH-ALL* algorithm, are similar to *Min-FINISH* and *Max-FINISH* respectively. They only differ with respect to the *Ready* set. In the greedy algorithms, the *Ready* set may contain subtasks from several levels. Initially, the *Ready* set contains all subtasks at level 1 from all applications. After mapping a subtask, the algorithms check if any of its successors are ready to be considered for scheduling and add them to *Ready* set. A subtask cannot be considered for scheduling until all its predecessors have been scheduled.

4. Results and Discussion

For the generalized resource scheduling problem considered above, it is not clear which variation of the list scheduling will perform best. Our intuition is that scheduling subtasks by considering all resource types together will result in bounded suboptimal solutions. In order to evaluate the effectiveness of the scheduling algorithms discussed in Sections 3.1 and 3.2, we have developed a software simulator that calculates the completion time for each of them. The input parameters are given to the simulator as fixed values or as a range of values with a minimum and maximum value.

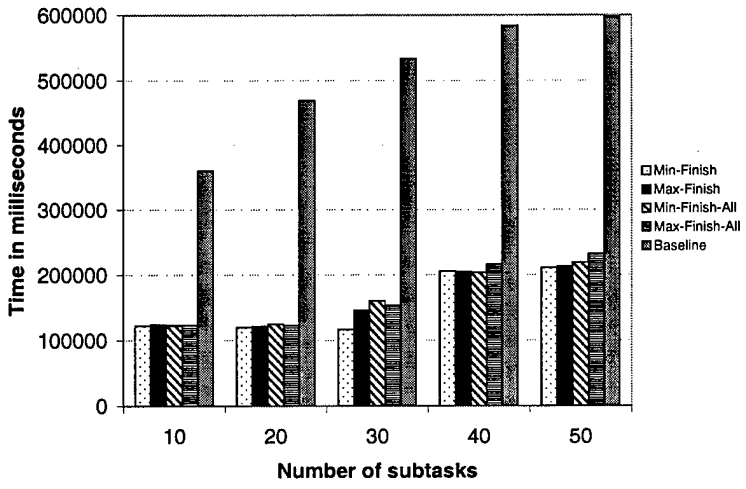


Figure 9. Simulation results for 20 machines and 6 data repositories with varying number of subtasks

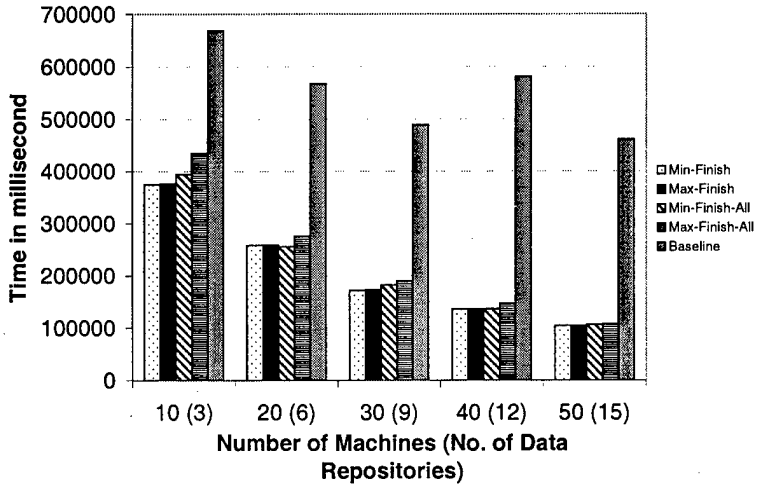


Figure 10. Simulation results for 50 subtasks with varying number of machines and data repositories

Subtask execution times, communication latencies, communication transfer rates, data items amounts, and data sets amounts, are specified to the simulator as range of values. The actual values of these parameters are choosen randomly by the simulator within the specified ranges. The fixed input parameters are the number of machines, the number of data repositories, the number of data items, and the total number of subtasks.

We assume that each task needs an input data set from the data repositories. This data set can be replicated and may be retreived from one or more data repositories. Each compute resource and data repository had several slots blocked at the beginning of the simulation to indicate advance reservations. We compare our scheduling algorithms with separated version of the baseline algorithm discussed in Section 2.4. The simulation results are shown in Figures 9 and 10. In Figure 9, the scheduling algorithms are compared for varying number of subtasks using 20 machines and 6 data repositories. Figure 10 shows a similar comparison for varying number of machines and data repositories with 50 subtasks. Our preliminary results show that all four of our heuristic algorithms seem to have similar performance with relatively uniform task costs. The simulation results clearly show that it is advantageous to schedule the system resources in a unified manner rather than scheduling each type of resource separately. Our scheduling algorithms have at least 30% improvment over the baseline algorithm which use the separated approach.

5. Future Work

This work represents, to the best of our knowledge, the first step towards a unified framework for resource scheduling with emerging constraints that are important in metacomputing. In this paper, we have considered one such requirement of advance reservations for compute resources and data repositories in this paper. We are investigating the question of how advance reservations impact task completion times. That is, in the scheduling, how soon we want to reserve a resource for a subtask to avoid waiting for another resource and/or blocking a different subtask. We are currently working on extending our scheduling algorithms to consider QoS requirements such as deadlines, priorities, and security. We are investigating the mapping of QoS specified at task level to subtasks in our framework.

In our future work we plan to develop scheduling algorithms for dynamic environments with the above mentioned resource requirements. In a dynamic environment, application tasks arrive in a real-time non-deterministic manner. System resources may be removed, or new resources may be added during run-time. Dynamic scheduling algorithms make use of real-time information and require feedback from the system.

References

[1] T. Adam, K. Chandy, and J. Dickson, "A comparison of list schedules for parallel processing systems," *Comm. of the ACM,* 17(12):685-690, Dec. 1974.

[2] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithm is independent of sizable variance in run-time predictions," *7th Heterogeneous Computing Workshop (HCW' 98),* pp. 79-87, March 1998.

[3] F. Berman and R. Wolski, "Scheduling from the perspective of the application," *5th IEEE International Symposium on High Performance Distributed Computing,* August 1996.

[4] F. Berman, "High-Performance schedulers," in *The Grid: blueprint for new computing infrastructure,* I. Foster and C. Kesselman, ed., Morgan Kaufmann Publishers, San Francisco, CA, 1999, pp. 279-309.

[5] T. Braun et al., "A Taxonomy for describing matching and scheduling heuristics for mixed-machines heterogeneous computing systems," *Workshop on Advances in Parallel and Distributed Systems (APADS),* West Lafayette, IN, Oct. 1998.

[6] T. Casavant and J. Kuhl, " A Taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. on Software Engineering,* 14(2):141-154, Feb. 1988.

[7] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Trans. on Software Engineering,* SE-15(11):1427-1436, Nov. 1989.

[8] I. Foster and C. Kesselman, ed., *The Grid: blueprint for new computing infrastructure,* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[9] R. Freund, B. Carter, D. Watson, E. Keith, and F. Mirabile, "Generational scheduling for heterogeneous computing systems," *Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '96),* pp. 769-778, Aug. 1996.

[10] R. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous computing environments with SmarNet," *7th Heterogeneous Computing Workshop (HCW '98),* pp. 184-199, March 1998.

[11] R. Freund, T. Kidd, D. Hensgen, and L. Moore, "SmartNet: a scheduling framework for heterogeneous computing," *The International Symposium on Parallel Architectures, Algorithms, and Networks,* Beijing, China, June 1996.

[12] R. Freund and H. J. Siegel, "Heterogeneous processing" *IEEE Computer,* 26(6):13-17, June 1993.

[13] Globus Web Page. *http://www.globus.org.*

[14] O. Ibarra and C. Kim, "Heuristic algorithms for scheduling independent tasks on non identical processors." *Journal of The ACM,* 24(2):280-289, April 1977.

[15] M. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," *7th Heterogeneous Computing Workshop (HCW' 98),* pp. 70-78, March 1998.

[16] M. Iverson, F. Ozguner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," *4th Heterogeneous Computing Workshop (HCW' 95),* pp. 93-100, Apr. 1995.

[17] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, " Heterogeneous computing: challenges and opportunities," *IEEE Computer,* 26(6):18-27, June 1993.

[18] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," *4th Heterogeneous Computing Workshop (HCW' 95),* pp. 30-34, Apr. 1995.

[19] Legion Web Page. *http://legion.virginia.edu.*

[20] M. Maheswaran and H. J. Siegel, "A Dynamic matching and scheduling algorithm for heterogeneous computing systems," *7th Heterogeneous Computing Workshop (HCW '98),* pp. 57-69,March 1998.

[21] R. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan, "Data-intensive computing," in *The Grid: blueprint for new computing infrastructure,* I. Foster and C. Kesselman, ed., Morgan Kaufmann Publishers, San Francisco, CA, 1999, pp. 105-129.

[22] MSHN Web Page. *http://www.mshn.org.*

[23] B. Shirazi, M. Wang, and G. Pathak, "Analysis and evaluation of heuristic methods for static task scheduling," *Journal of Parallel and Distributed Computing,* 10:222-232, 1990.

[24] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environment," *5th Heterogeneous Computing Workshop (HCW' 96),* pp. 98-117, Apr. 1996.

[25] G. C. Sih and E. A. Lee, "A Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans.on Parallel and Distributed Systems,* 4(2):175-187, Feb. 1993.

[26] L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM,* 35(6):45-52, June 1994.

[27] Lee Wang, Howard Jay Siegel, Vwani P. Roychowdhury, and Anthony A. Maciejewski, "Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach," *5Journal of Parallel and Distributed Computing,* 47(1):8-22, Nov. 1997.

Biographies

Ammar Alhusaini is a Ph.D. candidate in the Department of Electrical Engineering - Systems at the University of Southern California, Los Angeles, California, USA. His main research interest is task scheduling in heterogeneous environments. Mr. Alhusaini received a B.S. degree in computer engineering from Kuwait University in 1993 and M.S. degree in computer engineering from the University of Southern California in 1996. Mr. Alhusaini is a member of IEEE, IEEE Computer Society, and ACM.

Viktor K. Prasanna (V.K. Prasanna Kumar) is a Professor in the Department of Electrical Engineering - Systems, University of Southern California, Los Angeles. He received his B.S. in Electronics Engineering from the Bangalore University and his M.S. from the School of Automation, Indian Institute of Science. He obtained his Ph.D. in Computer Science from Pennsylvania State University in 1983. His research interests include parallel computation, computer architecture, VLSI computations, and high performance computing for signal and image processing, and vision. Dr. Prasanna has published extensively and consulted for industries in the above areas. He is widely known for his pioneering work in reconfigurable architectures and for his contributions in high performance computing for signal and image processing and image understanding. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He also serves on the editorial boards of the Journal of Parallel and Distributed Computing and IEEE Transactions on Computers. He is the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a Fellow of the IEEE.

Cauligi Raghavendra is a Senior Engineering Specialist in the Computer Science Research Department at the Aerospace Corporation. He received the Ph.D degree in Computer Science from University of California at Los Angeles in 1982. From September 1982 to December 1991 he was on the faculty of Electrical Engineering-Systems Department at University of Southern California, Los Angeles. From January 1992 to July 1997 he was the Boeing Centennial Chair Professor of Computer Engineering at the School of Electrical Engineering and Computer Science at the Washington State University in Pullman. He received the Presidential Young Investigator Award in 1985 and became an IEEE Fellow in 1997. He is a subject area editor for the Journal of Parallel and Distributed Computing, Editor-in-Chief for Special issues in a new journal called Cluster Computing, Baltzer Science Publishers, and is a program committee member for several networks related international conferences.

Session V

Invited Case Studies

Chair

Noe Lopez-Benitez
Texas Tech University

Metacomputing with MILAN

A. Baratloo P. Dasgupta* V. Karamcheti Z.M. Kedem
{baratloo,dasgupta,vijayk,kedem}@cs.nyu.edu
Courant Institute of Mathematical Sciences, New York University

Abstract

The MILAN project, a joint effort involving Arizona State University and New York University, has produced and validated fundamental techniques for the realization of efficient, reliable, predictable virtual machines, that is, metacomputers, on top of environments that consist of an unreliable and dynamically changing set of machines. In addition to the techniques, the principal outcomes of the project include three parallel programming systems—Calypso, Chime, and Charlotte—which enable applications be developed for ideal, shared memory, parallel machines to execute on distributed platforms that are subject to failures, slowdowns, and changing resource availability. The lessons learned from the MILAN project are being used to design Computing Communities, a metacomputing framework for general computations.

1. Motivation

MILAN (**M**etacomputing **I**n **L**arge **A**synchronous **N**etworks) is a joint project of Arizona State University and New York University. The primary objective of the MILAN project is to provide middleware layers that would enable the efficient, predictable execution of applications on an unreliable and dynamically changing set of machines. Such a middleware layer, will in effect create a *metacomputer*, that is a reliable stable platform for the execution of applications.

Improvements in networking hardware, communication software, distributed shared memory techniques, programming languages and their implementations have made it feasible to employ distributed collections of computers for executing a wide range of parallel applications. These "metacomputing environments," built from commodity machine nodes and connected using commodity interconnects, afford significant cost advantages in addition to their widespread availability (e.g., a machine on every desktop in an organization). However, such environments also present unique challenges for constructing metacomputers on them, because the component machines and networks may: (1) exhibit wide variations in performance and capacity, (2) become unavailable either partially or completely because of their use for other (non-metacomputing related) tasks. These challenges force parallel applications running on metacomputers to deal with an unreliable, dynamically changing set of machines and have thus, limited their use on all but the most decoupled of parallel computations.

As part of the MILAN project, we have been investigating fundamental techniques which would enable the effective use of metacomputing environments for a wide class of applications, originally concentrating on parallel ones. The key thrust of the project has been to develop run-time middleware that builds an *efficient, predictable, reliable virtual machine model* on top of unreliable and dynamically changing platforms. Such a virtual machine model would enable applications developed for idealized, reliable, homogeneous parallel machines to run unchanged on unreliable, heterogeneous metacomputing environments. Figure 1 shows the MILAN middleware in context. Our approach for realizing the virtual machine takes advantage of two general characteristics of computation behavior: *adaptivity* and *tunability*.

> **Adaptivity** refers to a flexibility in execution. Specifically, a computation is adaptive if it exhibits at least one of these two properties: (1) it can statically (at start time) and/or dynamically (during the execution) ask for resources satisfying certain characteristics and incorporate such resources when they are given to it, and (2) it can continue executing even when some resources are taken away from it.
>
> **Tunability** refers to a flexibility in specification. Specifically, a computation is tunable if it is able to trade off resource requirements over its lifetime, compensating for a smaller allocation of resources in one stage with a larger allocation in another stage and/or a change in the quality of output produced by the computation.

Our techniques leverage this flexibility in execution and specification to provide reliability, load balancing, and pre-

*On leave from Arizona State University

0-7695-0107-9/99 $10.00 © 1999 IEEE

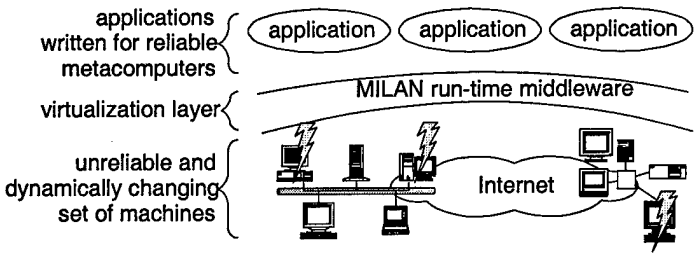


Figure 1. The MILAN middleware in context.

dictability, even when the underlying set of machines is unreliable and changing dynamically.

The principal outcomes of the MILAN project are (1) a core set of fundamental resource management techniques [21, 3, 23, 9] enabling construction of efficient, reliable, predictable virtual machines, and (2) the realization of these techniques in three complete programming systems: *Calypso* [3], *Chime* [27], and *Charlotte* [6]. Calypso extends C++ with parallel steps interleaved into a sequential program. Each parallel step specifies the independent execution of multiple concurrent tasks or a family of such tasks. Chime extends Calypso to provide nested parallel steps and inter-thread communication primitives (as expressed in the shared memory parallel language, Compositional C++ [8]). Charlotte provides a Calypso-like programming system and runtime environment for the Web. In addition to these systems, the MILAN project has also produced two general tools: *ResourceBroker* [4] and *KnittingFactory* [5], which support resource discovery and integration in distributed and web-based environments, respectively. More recently, as part of the *Computing Communities* project, we have been examining how the experience gained from designing, implementing, and evaluating these systems can be extended to supporting general applications on metacomputing environments.

The rest of the paper is organized as follows. Section 2 overviews the fundamental techniques central to all of the MILAN project's programming systems. The design, implementation, and performance of the various programming systems and general tools is described in detail in Section 3. Finally, Section 4 presents the rationale and preliminary design of Computing Communities, a metacomputing framework for general computations.

2 Key Techniques

To execute parallel programs on networks of commodity machines, one frequently assumes *a priori* knowledge—at program development—of the number, relative speeds, and the reliability of the machines involved in the computation. Using this information, the program can then distribute its load evenly for efficient execution. This knowledge can not be assumed for distributed multiuser environments, and hence, it is imperative that programs adapt to machine availability. That is, a program running on a metacomputer must be able to integrate new machines into a running computation, mask and remove failed machines, and balance the work load in such a way that slow machines do not dictate the progress of the computation.

The traditional solution to overcome this type of dynamically changing environment has been to write *self-scheduling* parallel programs (also referred to as the master/slave [16], the manager/worker [17], or the bag-of-tasks [7] programming model). In self-scheduled programs, the computation is divided into a large number of small computational units, or tasks. Participating machines pick up and execute a task, one at a time, until all tasks are done, enabling the computation to progress at a rate proportional to available resources. However, self scheduling does not solve all the problems associated with executing programs on distributed multiuser environments. First, self scheduling does not address machine and network failures. Second, a very slow machine can slow down the progress of faster machines if it picks up a compute-intensive task. Finally, self scheduling increases the number of tasks comprising a computation and, thereby, increases the effects of the overhead associated with assigning tasks to machines. Depending on the network, this overhead may be large and, in many cases, unpredictable.

The MILAN project extends the basic self-scheduling scheme in various ways to adequately address the above shortcomings. These extensions are embodied in five techniques: *eager scheduling*, *two-phase idempotent execution strategy* (TIES), *dynamic granularity management*, *preemptive scheduling*, and *predictable scheduling* for tunable computations. We describe the principal ideas behind each of these techniques here, deferring a detailed discussion of their implementation and impact on performance to the description of various programming systems in Section 3.

2.1 Eager Scheduling

Eager scheduling extends self scheduling to deal with network and machine failures, as well as any disparity in machine speeds. The key idea behind eager scheduling, initially proposed in [21], is that a single computation task can be concurrently worked upon by multiple machines. Eager scheduling works in a manner similar to self scheduling at the beginning of a parallel step, but once the number of remaining tasks goes below the number of available machines, eager scheduling aggressively assigns and re-assigns tasks until all tasks have been executed to completion. Concurrent assignment of tasks to multiple machines guarantees that slow machines, even very slow machines, do not slow down the computation. Furthermore, by considering failure as a special case of a slow machine (an infinitely slow machine), even if machines crash or become less accessible, for example due to network delays, the entire computation will finish as long as at least one machine is available for a sufficiently long period of time. Thus, eager scheduling masks machine failures without the need to actually detect failures.

2.2 Two-phase Idempotent Execution Strategy (TIES)

Multiple executions of a program fragment (which is possible under eager scheduling) can result in an incorrect program state. TIES [21] ensures *idempotent memory semantics* in the presence of multiple executions. The computation of each parallel step is divided into two phases. In the first phase, modifications to the shared data region, that is the write-set of tasks, are computed but kept aside in a buffer. The second phase begins when all tasks have executed to completion. Then, a single write-set for each completed task is applied to the shared data, thus atomically updating the memory. Note that each phase is idempotent, since its inputs and outputs are disjoint. Informally, in the first phase the input is shared data and the output is the buffer, and in the second phase the input is the buffer and the output is shared memory.

2.3 Dynamic Granularity Management

The interplay of eager scheduling and TIES addresses fault masking and load balancing. *Dynamic granularity management* (*bunching* for short) is used to amortize overheads and mask network latencies associated with the process of assigning tasks to machines. Bunching extends self scheduling by assigning a set of tasks (a bunch) as "a single assignment." Bunching has three benefits. First, it reduces the number of task assignments, and hence, the associated overhead. Second, it overlaps computation with communication by allowing machines to execute the next task (of a bunch) while the results of the previous task are being sent back on the network. Finally, bunching allows the programmer to write fine-grained parallel programs that are automatically and transparently executed in a coarse-grained manner.

We have implemented *factoring* [19], an algorithm that computes the bunch size based on the number of remaining tasks and the number of currently available machines.

2.4 Preemptive Scheduling

Eager scheduling provides load balancing and fault isolation in a dynamic environment. However, our description so far has considered only non-preemptive tasks which run to completion once assigned to a worker. Non-preemptive scheduling has the disadvantage of delivering sub-optimal performance when there is a mismatch between the set of tasks and the set of machines. Examples of situations include when the number of tasks is not divisible by the number of machines, when the tasks are of unequal lengths, and when the number of tasks is not static (i.e., new tasks are created and/or terminated on the fly). To address inefficiencies resulting from these situations, the MILAN project complements eager scheduling with preemptive scheduling techniques. Our results, discussed in Section 3, show that despite preemption overheads, use of preemptive scheduling on distributed platforms can improve execution time of parallel programs by reducing the number of tasks that need to be repeatedly executed by eager scheduling [23].

We have developed a family of preemptive algorithms, of which we present three here. The *Optimal Algorithm* is targeted for situations where the number of tasks to be executed is slightly larger than the number of machines available. This algorithm precomputes a schedule that minimizes the execution time and the number of context switches needed. However it requires that the task execution time be known in advance and therefore is not always practical. The *Distributed, Fault-tolerant Round Robin Algorithm* is suited for a set of n tasks scheduled on m machines, where $n > m$. Initially, the first m tasks are assigned to the m machines. Then, after a specified time quantum, all the tasks are preempted and the next m tasks are assigned. This continues in a circular fashion until all tasks are completed. The *Preemptive Task Bunching Algorithm* is applicable over a wider range of circumstances. All n tasks are bunched into m bunches and assigned to the m machines. When a machine finishes its assigned bunch, all the tasks on all the other machines are pre-empted and all the remaining tasks are collected, re-bunched (into m sets), and assigned again. This algorithm works well for both large-grained and fine-grained tasks even when machine speeds and task lengths vary.

2.5 Predictable Scheduling

While the techniques described earlier enable the building of an efficient, fault-tolerant virtual machine on top of an unreliable and dynamically changing set of machines, they alone are unable to address the predictability requirements of applications such as image recognition, virtual reality, and media processing that are increasingly running on metacomputers. One of the key challenges deals with providing sufficient resources to computations to enable them to meet their time deadlines in the face of changing resource availability.

Our technique [9] relies upon an explicit specification of application *tunability*, which refers to an application's ability to absorb and relinquish resources during its lifetime, possibly trading off resource requirements versus quality of its output. Tunability provides the freedom of choosing amongst multiple execution paths, each with their own resource allocation profile. Given such a specification and short-term knowledge about the availability of resources, the MILAN resource manager chooses an appropriate execution path in the computation that would allow the computation to meet its predictability requirements. In general, the resource manager will need to renegotiate both the level of resource allocation and the choice of execution path in response to changes in resource characteristics. Thus, application tunability increases its likelihood of achieving predictable behavior in a dynamic environment.

3. Programming Systems

We describe, in turn, three programming systems—Calypso, Chime, and Charlotte—which provide the concrete context in which the techniques described earlier have been implemented and evaluated. We then discuss the design and implementation of the ResourceBroker, a tool for dynamic resource association. KnittingFactory provides the same functionality as ResourceBroker, albeit for Web metacomputing environments, and is not being discussed because of space considerations.

3.1 Calypso

Commercial realities dictate that parallel computations typically will not be given a dedicated set of identical machines. Non-dedicated computing platforms suffer from non-uniform processing speeds, unpredictable behavior, and transient availability. These characteristics result from external factors that exist in "real" networks of machines. Unfortunately, load balancing, fault masking, and adaptive execution of programs on a set of dynamically changing machines are neglected by most programming systems. The neglect of these issues has complicated the already difficult job of developing parallel programs.

Calypso [3] is a parallel programming system and a runtime system designed for adaptive parallel computing on networks of machines. The work on Calypso has resulted in several original contributions which are summarized below.

Calypso separates the programming model from the execution environment: programs are written for a reliable virtual shared-memory computer with unbounded number of processors, i.e., a metacomputer, but execute on a network of dynamically changing machines. This presents the programmer with the illusion of a reliable machine for program development and verification. Furthermore, the separation allows programs to be parallelized based on the inherent properties of the problem they solve, rather than the execution environment.

Programs without any modifications can execute on a single machine, a multiprocessor, or a network of unreliable machines. The Calypso runtime system is able to adapt executing programs to use available resources—computations can dynamically scale up or down as machines become available, or unavailable. It uses TIES and allows parts of a computation executing on remote machines to fail, and possibly recover, at any point without affecting the correctness of the computation. Unlike other fault-tolerant systems, there is no significant additional overhead associated with this feature.

Calypso automatically distributes the work-load depending on the dynamics of participating machines, using eager scheduling and bunching. The result is that fine-grain computations are efficiently executed in coarse-grain fashion, and faster machines perform more of the computation than slower machines. Not only is there no cost associated with this feature, but it actually speeds up the computation, because fast machines are never blocked while waiting for slower machines to finish their work assignments—they bypass the slower machines. As a consequence, the use of slow machines will never be detrimental to the performance of a parallel program.

3.1.1 Calypso Programs

A Calypso program basically consists of the standard C++ programming language, augmented by four additional keywords to express parallelism. Parallelism is obtained by embedding *parallel steps* within sequential programs. Parallel steps consist of one or more task (referred to as *jobs* in the Calypso context), which (logically) execute in parallel and are generally responsible for computationally intensive segments of the program. The sequential parts of programs are referred to as *sequential steps* and they generally perform initialization, input/output, user interactions, etc.

Figure 2 illustrates the execution of a program with two parallel steps and three sequential steps. It is important to note that parallel programs are written for a virtual shared-

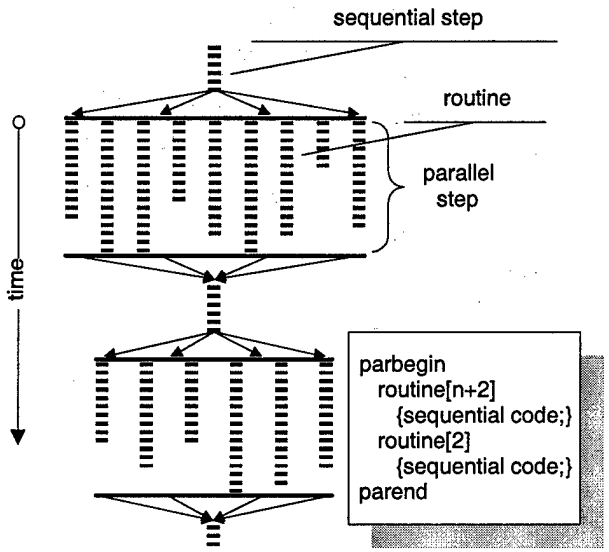


Figure 2. An execution of a program with two parallel steps and three sequential steps; the first parallel step consists of 9 jobs, the second parallel step consists of 6 jobs.

memory parallel machine irrespective of the number of machines that participate in a given execution.

This programming model is sometimes referred to as a block-structured parbegin/parend or fork/join model [13, 25]. Unlike other programming models where programs are decomposed (into several files or functions) for parallel execution, this model together with shared memory semantics, allows loop-level parallelization. As a result, given a working sequential program it is fairly straightforward to parallelize individual independent loops in an incremental fashion—if the semantics allows this.

Shared-memory semantics is only provided for *shared variables*, i.e., variables that are tagged with the `shared` keyword. A parallel step starts with the keyword `parbegin` and ends with the keyword `parend`. Within a parallel step, multiple parallel *jobs* may be defined using the keyword `routine`. Completion of a parallel step consists of completion of all its jobs in an indeterminate order.

3.1.2 Execution Overview

A typical execution of a Calypso program consists of a central process, called the *manager*, and one or more additional processes, called *workers*. These processes can reside on a single machine or they can be distributed on a network. In particular, when a user starts a Calypso program, in reality, she is starting a manager. Managers immediately fork a child process that executes as a worker.

The manager is responsible for the management of the computation as well as the execution of sequential steps. The current Calypso implementation only allows one manager, and therefore it does not tolerate the failure of this process. The computation of parallel jobs is left to the workers. In general, the number of workers and the resources they can devote to parallel computations can dynamically change in a completely arbitrary manner, and the program adapts to the available machines. In fact, the arbitrary slowdown of workers due to other executing programs on the same machine, failures due to process and machine crashes, and network inaccessibility due to network partitions are tolerated. Furthermore, workers can be added at any time to speed up an already executing system and to increase fault tolerance. Arbitrary slowdown of the manager is also tolerated; this would, of course, slow down the overall execution though.

3.1.3 Manager Process

The manager is responsible executing the non-parallel step of a computation as well as providing workers with *scheduling* and *memory* services.

Scheduling Service: Jobs are assigned to workers based on a self-scheduling policy. Moreover, the manager has the option of assigning a job repeatedly until it is executed to completion by at least one worker—this is eager scheduling, and provides the following benefits:

- As long as at least one worker does not fail continually, all jobs will be completed, if necessary, by this one worker.
- jobs assigned to workers that later failed are automatically reassigned to other workers; thus crash and network failures are tolerated.
- Because workers on fast machines can re-execute jobs that were assigned to slow machines, they can bypass a slow worker to avoid delaying the progress of the program.

In addition to eager scheduling, Calypso's scheduling service implements several other scheduling techniques for improved performance. Bunching masks network latencies associated with the process of assigning jobs to workers.It is implemented by sending the worker a range of job Ids in each assignment. The overhead associated with this implementation is one extra integer value per job assignment message, which is negligible.

Memory Service: Since multiple executions of jobs caused by eager scheduling may lead to an inconsistent memory state, managers implements TIES as follows. Before each parallel step, a manager creates a twin copy of the shared pages and unprotects the shared region. The memory management service then waits until a worker either requests a

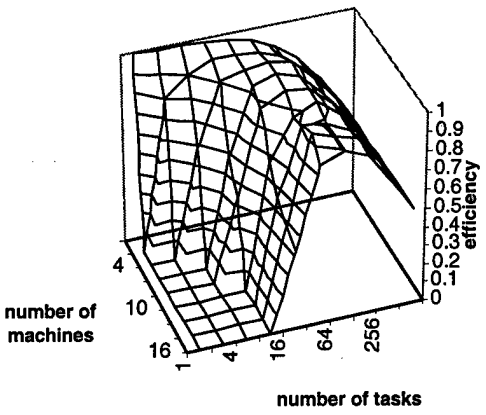
page or reports the completion of a job. The manager uses the twin copy of the shared pages to service worker page requests. The message that workers send to the manager to report the completion of a job also contains the modifications that resulted from executing the job. Specifically, workers logically bit-wise XORs the modified shared pages before and after executing the job, and send the results (diffs) to the manager. When a manager receives such a message, it first checks whether the job has been completed by another worker. If so, the diffs are discarded, otherwise, the diffs are applied (by an XOR operation) to manager's memory space. Notice that the twin copies of the shared pages, which are used to service worker page requests, are not modified. The memory management of a parallel step halts once all the jobs have run to completion, and the program execution then continues with the next sequential step.

3.1.4 Worker Process

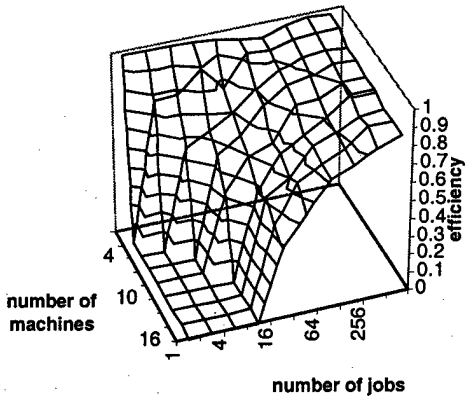
A worker repeatedly contacts the manager for jobs to execute. The manager sends the worker an assignment (a bunch of jobs) specified by the following parameters: the address of the function, the number of instances of the job, and a range of job Ids. After receiving a work assignment, a worker first access-protects the shared pages, and then calls the function that represents an assigned job. The worker handles page-faults by fetching the appropriate page from the manager, installing process' address space, and unprotecting the page so that subsequent accesses to the same page will proceed undisturbed. Once the execution of the function (i.e. the job) completes, the worker identifies all the modified shared pages and sends the diffs to the manager and starts executing the next job in the assignment. Notice that bunching overlaps computation with communication by allowing a worker to execute the next job while the diffs are on the network heading to the manager.

Additional optimizations have been implemented, including the following:

Caching: For each shared page, the manager keeps track of the logical step-number in which the page was last modified. This vector is piggybacked on a job assignment the first time a worker is assigned a job in a new parallel step. Hence, the associated network overhead is negligible. Workers use this vector on page-faults to *locally determine* whether the cached copy of a page is still valid. Thus, pages that have paged-in by workers are kept valid as long as possible without a need for an invalidation protocol. Modified shared pages are re-fetched only when necessary. Furthermore, read-only shared pages are fetched by a worker at most once and write-only shared pages are never fetched. As a result, programmer does not declare the type of coherence or caching technique to use, rather, the system dynamically adapts. Invalidation requests are piggybacked on work assignment messages and bear very little additional cost.



(a) PVM



(b) Calypso

Figure 3. Parallel ray tracing with different number of parallel tasks.

Prefetching: Prefetching refers to obtaining a portion of the data before it is needed, in the hope that it will be required sometime in the future. Prefetching has been used in a variety of systems with positive results. A Calypso worker implements prefetching by monitoring its own data access patterns and page-faults, and it tries to predict future data access based on past history. The predictions are then used to pre-request shared pages from the manager. Depending on the regularity of a program's data access patterns, prefetching has shown positive results.

3.1.5 Performance Experiments

The experiments were conducted on up to 17 identical 200 MHz PentiumPro machines running Linux version 2.0.34 operating system, and connected by a 100Mbps Ethernet through a non-switched hub. The network was isolated to eliminate outside effects.

A publicly available sequential ray tracing program [10] was used as the starting point to implement parallel versions in Calypso and PVM [16]. The sequential program, which traced a 512×512 image in in 53 s, is used for calculating the parallel efficiencies.

The PVM implementation used explicit master/slave programming style for load balancing, where as for Calypso, load balancing was provided transparently by the run-time system. To demonstrate the effects of adaptivity, the PVM and Calypso programs were parallelized using different number of tasks and executed from 1 to 16 machines. The performance results are illustrated in Figure 3. As the results indicate, the PVM program is very sensitive to the number and the computation requirement of the parallel tasks, and at most, a hand-tuned PVM program outperforms a Calypso program by 4%. Notice that independent of the number of machines used, the interplay of bunching, eager scheduling, and TIES allows the Calypso program to achieves its peak performance using 512 tasks—fine grain tasks: as the result of bunching, fine-grain tasks, in effect, execute in coarse-grain fashion; the combination of eager scheduling and TIES compensates any over-bunching that may occur.

3.2 Chime

Chime is a parallel processing system that retains the salient features of Calypso, but supports a far richer set of programming features. The internals of Chime are significantly different from Calypso, and it runs on the Windows NT operating system [27]. Chime is the first system that provides a true general shared memory multiprocessor environment on a network of machines. It achieves this by implementing the CC++ [8] language (shared memory part) on a distributed system. Thus in addition to Calypso features of fault-tolerance and load balancing Chime provides:

- True multi-processor shared-memory semantics on a network of machines.
- Block structured scoping of variables and non-isolated distributed parallel execution.
- Support for nested parallelism.
- Inter-task synchronization.

3.2.1 Chime Architecture

A program written in CC+ is preprocessed to convert it to C++ and compiled and linked with the Chime library. Then the executable is run, using the manager-worker scheme of Calypso.

The manager process consists of two threads, the *application thread* and the *control thread*. The application thread executes the code programmed by the programmer. The control thread executes, exclusively, the code provided by the Chime library. Hence, the application thread runs the program and the control thread runs the management routines, such as scheduling, memory service, stack management, and synchronization handling.

The worker process also consists of two threads, the application thread and the control thread. The application threads in the worker and manager are identical. However, the control thread in the worker is the client of the control thread in the manager. It requests work from the manager, retrieves data pages from the manager and flushes updated memory to the manager at the end of the task execution.

3.2.2 Chime and CC++

As mentioned earlier, Chime provides a programming interface that is based on the Compositional C++ or CC++ [8] language definition. CC++ provides language constructs for shared memory, nested parallelism and synchronization. All threads of the parallel computation share all global variables. Variables declared local to a function are private to the thread running the function, but if this thread creates more threads inside the function, then all the children share the local variables.

CC++ uses the *par* and *parfor* statements to express parallelism. Par and parfor statements can be nested. CC++ uses single assignment variables for synchronization. A single assignment variable is assigned a value by any thread called the writing thread. Any other thread, called the reading thread can read the written value. The constraint is that the writing thread has to assign before the reading thread reads, else the reading thread is blocked until the writing thread assigns the variable.

These language constructs provide significant challenges to a distributed (DSM-based) implementation that is also fault tolerant. We achieved the implementation by using a pre-processor to detect the shared variables and parallel constructs, providing stack-sharing support—called distributed cactus stacks—to implement parent-child variable sharing and innovative scheduling techniques, coupled with appropriate memory flushing to provide synchronization [28].

3.2.3 Preprocessing CC++

Consider the following parallel statement:

```
parfor ( int i=0; i<100; i++) {
     a[i] = 0;
};
```

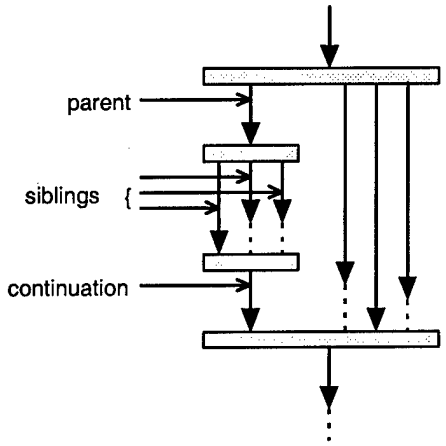


Figure 4. A DAG for a nested parallel step.

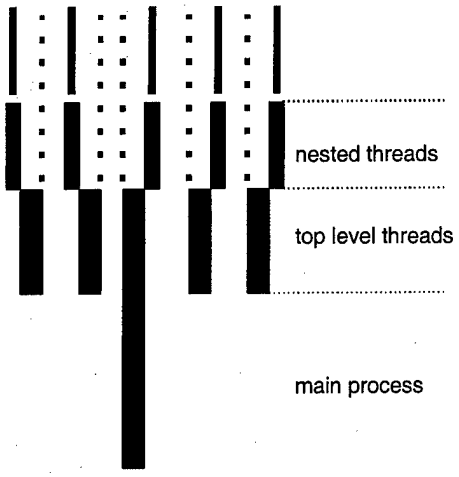


Figure 5. Graphical representation of a cactus stack.

This creates 100 tasks, each task assigning one element of the array a. The preprocessor converts the above statement to something along the following lines:

```
1.   for (int i=0; i<100;i++) {
2.      add task entry and &i
        in the scheduling table;
     }
3.   SaveContext of this thread;
4.   if worker {
        a[i] = 0;
5.      terminate task;
     }
6.   else {
7.      suspend this thread and
        request manager to
        schedule threads till
        all tasks completed;
     }
```

The above code may execute in the manager (top level parallelism) or the worker (nested parallelism). *Assume the above code executes in the manager.* Then the application thread of the manager executes the code. Lines 1 and 2 create 100 entries in the scheduling table, one per parallel task. Then line 3 saves the context of the parent task, including the parent stack. Then the parent moves to line 7 and this causes the application thread to transfer control to the control thread.

The control thread now waits for task assignment requests from the control threads of workers. When a worker requests a task, the manager control thread sends the stored context and the index value of i for a particular task to the worker.

The control thread in the worker installs the received context and the stack on the application thread in the worker and resumes the application thread. This thread now starts executing at line 4. Note that now the worker is executing at line 4, and hence does one iteration of the loop and terminates. Upon termination, the worker control thread regains control, flushes the updated memory to the manager and asks the manger for a new assignment.

3.2.4 Scheduling

The controlling thread at the manager is also responsible for task assignment, or scheduling. The manager uses a scheduling algorithm that takes care of task allocation to the workers as well as scheduling of nested parallel tasks in correct order. Nested parallel tasks in an application form a DAG as shown in Figure 4.

Each nested parallel step consists of several *sibling* parallel tasks. It also has a parent task and a continuation that must be executed, once the nested parallel step has been completed. A continuation is an object that fully describes a future computation. To complicate the scenario, a continuation may itself have nested parallel step(s).

The manager maintains an execution dependency graph to capture the dependencies between the parallel tasks and schedules them and their corresponding continuations in correct order. Eager scheduling is used to allocate tasks to the workers.

3.2.5 Cactus Stacks

The cactus stacks are used to handle sharing of local variables (see Figure 5). For top level nesting, the manager process is suspended at a point in execution where its stack and context should be inherited by all the children threads. When a worker starts, it is sent the contents of the manager's stack along with the context. The controlling thread of the worker process then installs this context as well as the stack, and starts the application thread.

However, if a worker executes a nested parallel step, the same code as the above case is used, but the runtime system behaves slightly differently. The worker, after generating the nested parallel jobs, invokes a routine that adds the jobs and the continuation of the parent job to the manager's job table, remotely. The worker suspends and the controlling thread in the worker, sends the worker's complete context, including the newly grown stack, to the manager.

The stack for a nested parallel task, therefore, is constructed by writing the stack segments of its ancestors onto the stack of a worker's application thread. Upon completion, the local portion of the stack for a nested parallel task is unwound leaving only those portions that represent its ancestors. This portion of the stack is then XOR'ed with its unmodified shadow and the result is returned to the manager.

3.2.6 Performance Experiments

Many performance tests have been done on Chime [27], evaluating its capabilities in speedups, load balancing, and fault tolerance. The results are competitive to other systems, including Calypso. We present three micro-tests that show the performance of the nested parallelism (including cactus stacks), the Chime synchronization mechanisms, and preemptive scheduling mechanisms.

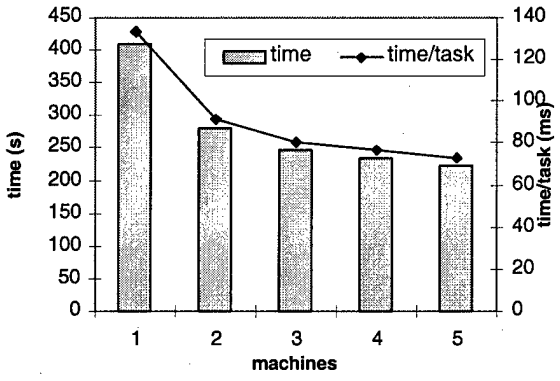


Figure 6. Performance of Nested Parallelism.

For the nested parallelism overhead, we run a program that recursively creates two child threads until 1024 leaf threads have been created. Each leaf thread assigns one integer in a shared array and then terminates. Figure 6 shows that the total runtime of the program asymptotically saturates as number of machines are increased, due to the bottleneck in stack and thread management at the manager. The time taken to handle all overhead for a thread (including cactus stacks) is 74 ms.

To measure the synchronization overhead, we use 512 single assignment variables, assign them from 512 threads and read them from 512 other threads. As can be seen in

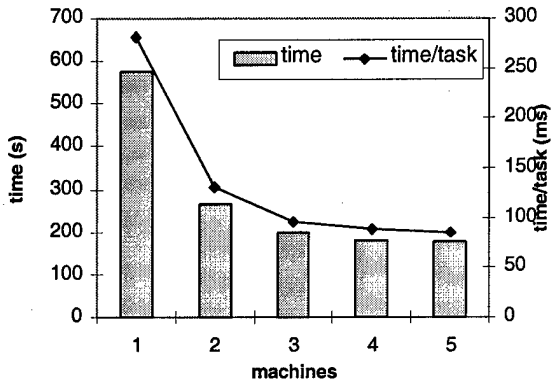


Figure 7. Performance of Synchronization.

Figure 7, the synchronization overhead is about 86 ms per occurrence, showing that synchronization does not add too much overhead over basic thread creation.

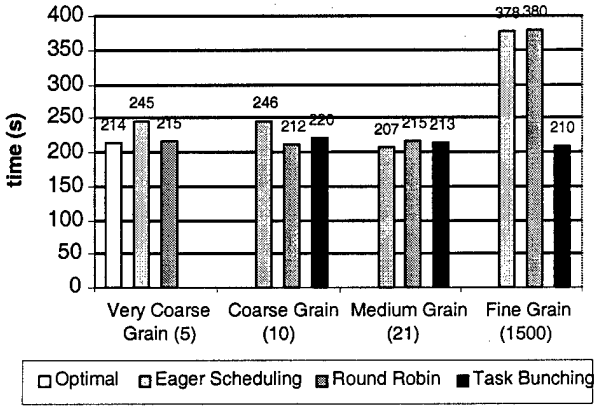


Figure 8. Performance of Preemptive Scheduling.

To measure the impact of preemptive scheduling algorithms for programs with different grain sizes, we decomposed a matrix-multiply algorithm on two 1500 × 1500 matrices into 5 tasks (very coarse grain), 10 tasks (coarse grain), 21 tasks (medium grain), and 1500 tasks (fine grain). All experiments used three identical machines. Given the equal task lengths, our experiments were biased against preemptive schedulers. As shown in Figure 8, on the overall, preemptive scheduling has definite advantages over non-preemptive scheduling, not withstanding its additional overheads. Specifically, for coarse-grained and very coarse-grained tasks, round robin scheduling effectively complements eager scheduling in reducing overall execution time. For most other task sizes, the preemptive task bunching algorithm yields the best performance; for fine-grained tasks

it minimizes the number of preemptions that are necessary.

3.3 Charlotte

Many of the assumptions made for (local-area) networks of machines are not valid for the Web. For example, the machines on the Web do not have a common shared file system, no single individual has access-rights (user-account) on every machine, and the machines are not homogeneous. Another important distinction is the concept of *users*. A user who wants to execute a program on a network of machines, typically performs the steps: logs onto a machine under her control (i.e. the *local* machine), from the local machine logs onto other machines on the network (i.e. *remote* machines) and initializes the execution environment, and then starts the program. In the case of the Web, no user can possibly hope to have the ability to log onto remote machines. Thus, another set of users who control remote machines, or software agents acting on their behalf, must voluntarily allow others access. To distinguish the two types of users, this section uses the term *end-users* to refer to individuals who start the execution (on their local machines) and await results, and *volunteers* to refer to individuals who voluntarily run parts of end-users' programs on their machines (remote to end-users). Similarly, *volunteer machines* is used to refer to machines owned by volunteers.

Simplicity and security are important objectives for volunteers. Unless the process of volunteering a machine is simple—for example as simple as a single mouse-click—and the process of withdrawing a machine is simple, it is likely that many would-be volunteer machines will be left idle. Furthermore, volunteers need assurance that the integrity of their machine and file system will not be compromised by allowing "strangers" to execute computations on their machines. Without such an assurance, it is natural to assume security concerns will outweigh the charitable willingness volunteering.

Charlotte [6] is the *first parallel programming system to provide one-click computing*. The idea behind one click computing is to allow volunteers from anywhere on the Web, and without any administrative effort, to participate in ongoing computations by simply directing a standard Java-capable browser to a Web site. A key ingredient in one-click computing is its lack of requirements: user-accounts are not required, the availability of the program on a volunteer's machine is not assumed, and system-administration is not required. Charlotte builds on the capability of the growing number of Web browsers to seamlessly load Java applets from remote sites, and the applet security model, which enables Web browsers to execute untrusted applets in a controlled environment, to provide a comprehensive programming system.

3.3.1 Charlotte Programs

A Charlotte program is written by inserting any number of *parallel steps* onto a sequential Java program. A parallel step is composed of one or more *routines*, which are (sequential) threads of control capable of executing on remote machines.

A parallel step starts and ends with the invocation of `parBegin()` and `parEnd()` methods, respectively. A routine is written by subclassing the `Droutine` class and overriding its `drum()` method. Routines are specified by invoking the `addRoutine()` method with two arguments: a routine object and an integer, n, representing the number of routine instances to execute. To execute a routine, the Charlotte runtime system invokes the `drun()` method of routine objects, and passes as arguments the number of routine instances created (i.e. n) and an identifier in the range $(0, \ldots, n]$ representing the current instance.

A program's data is logically partitioned into *private* and *shared* segments. Private data is local to a routine and is not visible to other routines; shared data, which consists of *shared class-type* objects, is distributed and is visible to all routines. For every basic data-type defined in Java, Charlotte implements a corresponding distributed shared class-type. For example, Java provides `int` and `float` data-types, whereas Charlotte provides `Dint` and `Dfloat` classes. The class-types are implemented as standard Java classes, and are read and written by invoking `get()` and `set()` method calls, respectively. The runtime system maintains the coherence of shared data.

3.3.2 Implementation

Worker Process: A Charlotte worker process is implemented by the `Cdaemon` class which can run either as a Java application or as a Java applet. At instantiation, a `Cdaemon` object establishes a TCP/IP connection to the manager and maintains this connection throughout the computation.

Two implementation features are worth noting. First, since `Cdaemon` is implemented as an applet (as well as an application), the code does not need to be present on volunteer machines before the computation starts. By simply embedding the `Cdaemon` applet in an HTML page, browsers can download and execute the worker code. Second, the `Cdaemon` class, unlike its counterpart the Calypso worker, is independent of the Charlotte program it executes. Thus, not only are Charlotte workers able to execute parallel routines of any Charlotte program, but only the necessary code segments are transfered to volunteer machines.

Manager Process: A manager process begins with the `main()` method of a program and executes the non-parallel steps in a sequential fashion. It also manages the progress of parallel steps by providing scheduling and memory services

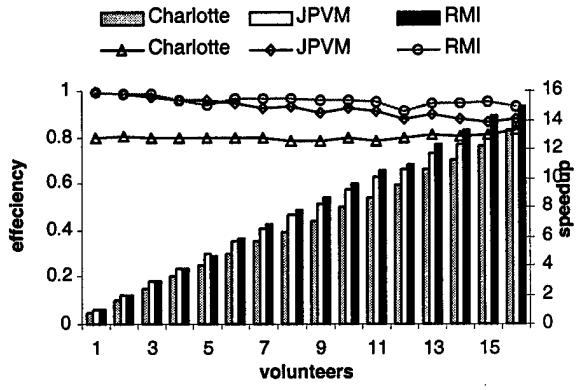


Figure 9. Performance comparison of Charlotte, RMI and JPVM programs.

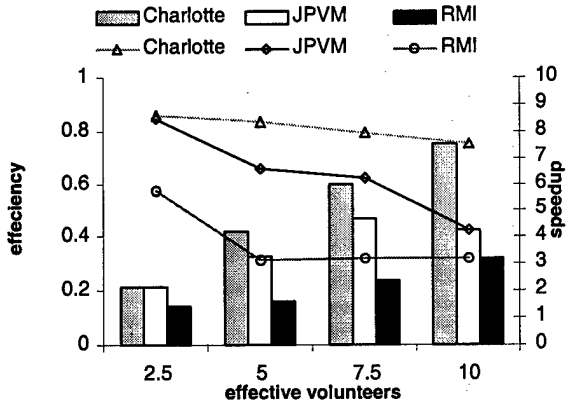


Figure 10. Load balancing of Charlotte, RMI and JPVM programs.

to workers. They are based on eager scheduling, bunching, and TIES.

Distributed Shared Class Types: Charlotte's distributed shared memory is implemented in pure Java at the data-type level; that is, through Java classes as stated above. For each primitive Java type like `int` and `float`, there is a corresponding Charlotte class-type `Dint` and `Dfloat`. The member variables of these classes are a `value` field of the corresponding primitive type, and a `state` flag that can be `not_valid`, `readable`, or `dirty`. It is important to note that different parts of the shared data can be updated by different worker processes without false sharing, as long as the CRCW-Common condition is met. (That is, several workers in a step can update the same data element, as long as all of them write the same value.) The shared memory is always logically coherent, independently of the order in which routines are executed.

3.3.3 Performance Experiments

The experiments were conducted in the same execution environment as in Section 3.1.5. Programs were compiled (with compiler optimization turned on) and executed in the Java Virtual Machine (JVM) packaged with Linux JDK 1.1.5 v7. TYA version 0.07 [22] provided just-in-time compilation.

A publicly available sequential ray tracing program [24] was used as the starting point to implement parallel versions in Charlotte, Java RMI [14], and JPVM [15]. Java RMI is an integral part of Java 1.1 standard and, therefore, it is a natural choice for comparison. JPVM is a Java implementation of PVM, one of the most widely used parallel programming systems. For the experiments, a 500 × 500 image was traced. The sequential program took 154 s to complete, and this number is used as the base in calculating the speedups.

The first series of experiments compares the performance of the three parallel implementations of ray tracer, see Figure 9. In the case of Charlotte, the same program with the same runtime arguments was used for every run—the program tuned itself to the execution environment. For RMI and JPVM programs, on the other hand, executions with different grain sizes were timed and the best results are reported—the programs were hand-tuned for the execution environment. The results indicate that when using 16 volunteers, the Charlotte implementation runs within 5% and 10% of hand-tuned JPVM and RMI implementations, respectively. It is encouraging to see that the performance of Charlotte is competitive with other systems that do not provide load balancing and fault masking.

The final set of experiments illustrates the efficiency of the programs when executing on machines of varying speeds—a common scenario when executing programs on the Web. Exactly the same programs with the same granularity sizes as the previous experiment were run on n, $1 \leq n \leq 4$, groups of volunteers, where each group consisted of four machines: one normal machine, one machine slowed down by 25%, one machine slowed down by 50%, and one machine slowed down by 75%. Each group has a computing potential of 2.5 volunteer machines. The results are depicted in Figure 10. As the results indicate, the Charlotte program is the only one able to maintain its efficiency—the efficiency of the Charlotte program degraded by approximately 5%. In contrast, the efficiency of RMI and PVM programs dropped by as much as 60% and 45%, respectively.

3.4 ResourceBroker

ResourceBroker [4] is a resource management system for monitoring computing resources in a distributed mul-

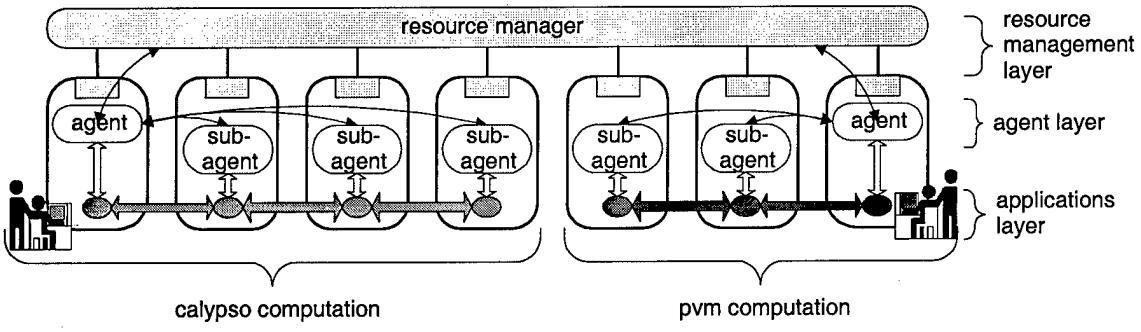


Figure 11. The components of ResourceBroker that comprise of the resource management and the agent layers.

tiuser environments and for dynamically assigning them to concurrently executing computations. Although applicable to a wide variety of computations, including sequential ones, it especially benefits adaptive parallel computations.

Adaptive parallel computations can effectively use networked machines because they dynamically expand as machines become available and dynamically acquire machines as needed. While most parallel programming systems provide the means to develop adaptive programs, they do not provide any functional interface to external resource management systems. Thus, no existing resource management system has the capability to manage resources on commodity system software, arbitrating the demands of multiple adaptive computations written using diverse programming environments. Indeed, existing resource management systems are tightly integrated with the programming system they support and their inability to support more than one programming system severely limits their applicability.

ResourceBroker is built to validate a set of novel mechanisms that facilitate dynamic allocation of resources to adaptive parallel computations. The mechanisms utilize low-level features common to many programming systems, and unique in their ability to transparently manage *adaptive parallel programs that were not developed to have their resources managed by external systems*. The ResourceBroker prototype is the first system that can support adaptive programs written in more than one programming system, and has been tested using a mix of programs written in PVM, MPI, Calypso, and PLinda.

4 Computing Communities: Metacomputing for General Computations

So far, we have addressed metacomputing for parallel computations. Operating systems such as Amoeba [29], Plan-9 [26], Clouds [12] and to an extent Mach [1] had targeted the use of distributed systems for seamless general purpose computing. However, the rise of commodity operating systems and the need for application binary compatibility have made such approaches less attractive, necessitating instead that general computations also be supported on metacomputing environments. To enable the latter, we have designed and will implement the *Computing Community* (CC) framework.

A Computing Community (CC) is a collection of machines (with dynamic membership) that form a single, dynamically changing, virtual multiprocessor system. It has global resource management, dynamic (automatic) reconfigurability, and the ability to run binaries of all applications designed for a base operating system. The physical network disappears from the view of the computations that run on the CC.

The CC brings flexibility of well-designed, distributed computing environments to the world of non-distributed applications-including legacy applications-without the need for distributed programming, new APIs, RPCs, object-brokerage, or similar mechanisms.

4.1 Realizing a CC

We are in the process of building a CC on top of the Windows NT operating system, with the initial software architecture shown in Figure 12. The CC comprises three synergistic components: (1) Virtual Operating System (2) Global Resource Manager (3) Application Adaptation System.

The *Virtual Operating System* (VOS) is a layer of software that non-intrusively operates between the applications and the standard operating system. The VOS presents the standard Windows NT API to the application, but can execute the same API calls differently, thereby extending the OS's power. The VOS essentially decouples the virtual entities required for executing a computation from their mappings to physical resources in the CC.

The *Global Resource Manager* manages all CC resources, dynamically discovering the availability of new re-

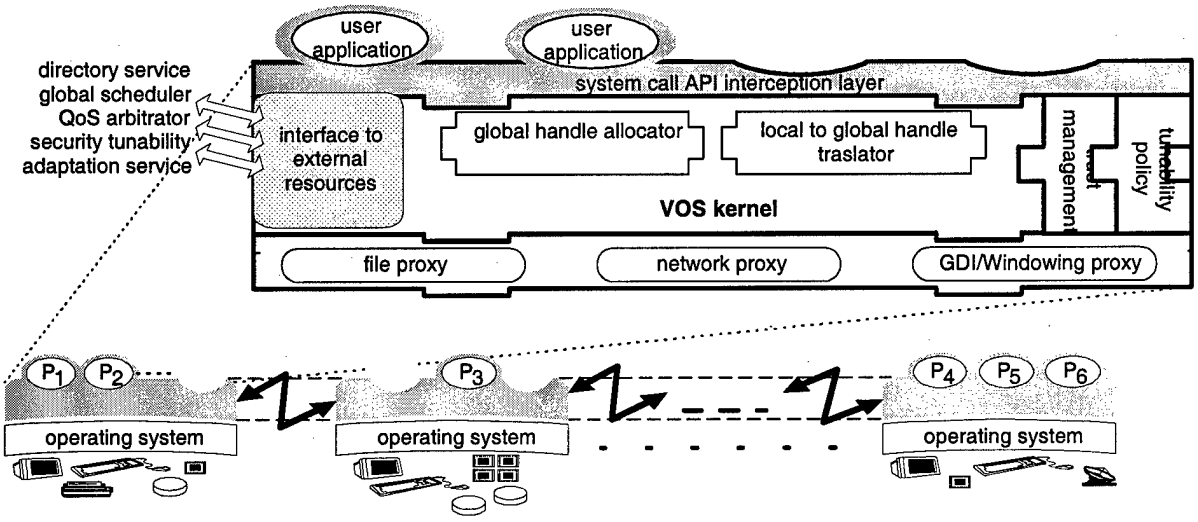


Figure 12. Software architecture for Computing Communities.

sources, integrating them into the CC, and making them available for use by CC computations. It handles resource requests from other components of the system and satisfies them as per scheduling requirements.

The *Application Adaptation System* enables the computations to take full advantage of CC resources and provides dynamic reconfiguration capabilities. Adaptation techniques allow computations to become aware of and gracefully adapt themselves to changes in CC resource characteristics.

Figure 13 shows a conceptual view of a CC. It takes a set of operating systems, and a set of resources, and via a layer of middleware converts it into an integrated community. CCs can expand and contract dynamically, and the computations are completely mobile within CCs. In short, using the CC framework, the computation transparently acquires the benefit of operating in a distributed environment.

4.2 The Virtualization Concept

Under a standard OS, a process runs in a logical address space, is bound to a machine, and interacts with the OS local to this machine. In fact, the processes (and their threads) are virtualizations of the real CPUs. However, such virtualization is low-level and limited in scope.

In the CC, virtualization is defined at a much higher level, and all physical resources (CPU, memory, disks, and networks) as well as the OSs on to all the machines are aggregated into a single, unified (distributed) virtual resource space.

A process in the CC is enveloped in a virtual shell (Figure 14), which makes the process feel that it is running on a standard OS. However, the shell creates a virtual world made of the aggregate of the physical worlds in the CC.

Consider a user U who starts an application A (and its GUI) on some machine M_1. Soon, U abruptly moves to another machine M_2. Now U can instruct the CC to connect the virtual screen, virtual keyboard, and the virtual mouse of A to the physical resources of M_2. The CC complies and U continues working on M_2, as if A executed there. Later the CC might decide it preferable to run the application on M_2. The scheduler then transparently moves A to M_2 preserving process state and open files and network connections.

The above simple scenario shows a particular aspect of the power of virtualization. In general:

- The users can move their virtual "home machines" at will, even for applications that are currently executing. This is the ultimate mobile computing scenario.
- A critical service running on machine M_1 can be moved to machine M_2 if M_1 has to be relinquished.
- Schedulers can control the complete set of resources.
- The provision of multiple physical resources for a single virtual resource delivers important new capabilities ranging from duplicating application displays on multiple screens to replicating processes for fault tolerance.

The CC functionality relies upon three key mechanisms: API interception, proxies, and translations between physical and logical handles. API interception allows the API calls from an application to the operating system to be intercepted and the behavior of the API call to be modified. After intercepting a call, the virtual operating system (VOS) does one of the following operations. (1) Passes the call on to the local Windows NT operating system. (2) Passes the call to a remote Windows NT operating system. (3) Executes the call inside the VOS. (4) Executes some VOS code

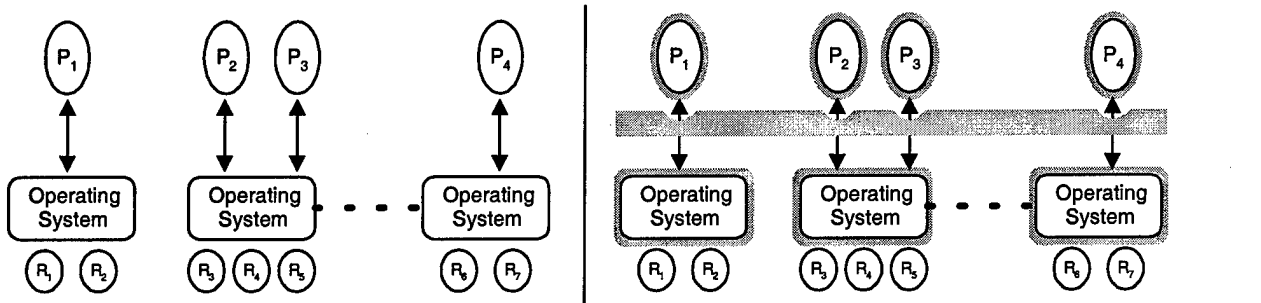


Figure 13. A conceptual view of Computing Communities.

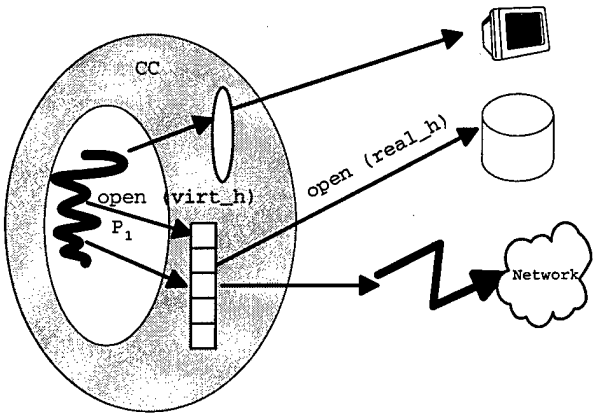


Figure 14. Virtualization of a process.

and then passes the call to a local or remote Windows NT system.

In order to reallocate processes to machines, a general form of process migration is necessary. To move a process from one location to another, just moving the state is not enough, all connections and handles have to be moved. This can be achieved by having proxies that emulate the connections of the process after the process has moved. For example, if a process P moving from M_1 to M_2 has an open networking connection to M_3, a proxy is created on M_1, which keeps the original connection to M_3 open, and then forwards messages between P and M_3, after P has moved.

Equally essential to successful virtualization of resources for migrating processes is the use of virtual handles. For example when a process opens a file on top of a VOS, the VOS intercepts this call and stores the returned physical handle but returns to the process a handle, which we refer to as virtual. The virtual handle can be used by the process, regardless of migrations, to access that file, due to the transparent translation service provided by the VOS. The virtual handles are used to virtualize I/O connections, sub-processes, threads, files, network sockets, etc.

Acknowledgements

This paper describes work of not only the authors but also other participants in the MILAN project, particularly Fangzhe Chang, Ayal Itzkovitz, Mehmet Karaul, Holger Karl, Donald McLaughlin, Shantanu Sardesi, Peter Wyckoff, and Yuanyuan Zhao. This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grants number CCR-94-11590 and CCR-95-05519; by Deutsche Forschunggemeinschaft; by Intel; and by Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. *Proceedings Summer USENIX*, July 1986.

[2] Y. Aumann, Z. M. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proceedings of the Annual Symposium on the Foundations of Computer Science (FOCS)*, 1993.

[3] A. Baratloo, P. Dasgupta, and Z. M. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of International Symposium on High-Performance Distributed Computing (HPDC)*, 1995.

[4] A. Baratloo, A. Itzkovitz, Z. M. Kedem, and Y. Zhao. Mechanism for just-in-time allocation of resources to adaptive parallel programs. In *Proceedings of International Parallel Processing Symposium (IPPS/SPDP)*, 1999.

[5] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. KnittingFactory: An infrastructure for distributed web applications. *Concurrency: Practice and Experience*, 10(11–13):1029–1041, 1998.

[6] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1996.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.

[8] M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 1992.

[9] F. Chang, V. Karamcheti, and Z. M. Kedem. Exploiting application tunability for efficient, predictable parallel resource management. In *Proceedings of International Parallel Processing Symposium (IPPS/SPDP)*, 1999.

[10] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[11] P. Dasgupta, Z. M. Kedem, and M. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of 15th International Conference on Distributed Computing Systems (ICDCS)*, 1995.

[12] P. Dasgupta, R. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 1991.

[13] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, Sept. 1965.

[14] T. Downing. *Java RMI: Remote Method Invocation.* IDG Books Worldwide, 1998.

[15] A. Ferrari. JPVM—network parallel computing in Java. In *Proceedings of the Workshop on Java for High-Performance Network Computing*, 1998.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine.* MIT Press, 1994.

[17] W. Gropp, E. Lust, and A. Skjellum. *Using MPI: Portable parallel programming with the message-passing interface.* MIT Press, 1994.

[18] S. Huang and Z. M. Kedem. Supporting a flexible parallel programming model on a network of workstations. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, 1996.

[19] S. F. Hummel, E. Edith Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, Aug. 1992.

[20] Z. M. Kedem and K. Palem. Transformations for the automatic derivation of resilient parallel programs. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1992.

[21] Z. M. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 1990.

[22] A. Kleine. Tya archive. Availabe at `http://www.dragon1.net/software/tya`.

[23] D. McLaughlin. *Scheduling Fault-tolerant, Parallel Computations in a Distributed Environment.* PhD thesis, Arizona State University, December 1997.

[24] L. McMillan. An instructional ray-tracing renderer written for UNC COMP 136 Fall '96. Available at `http://graphics.lcs.mit.edu/~capps/iap/class3/RayTracing/RayTrace.java`, 1996.

[25] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 1997.

[26] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell labs. In USENIX, editor, *Computing Systems, Summer, 1995.*, volume 8, pages 221–254, Berkeley, CA, USA, Summer 1995. USENIX.

[27] S. Sardesai. *Chime: A Versatile Distributed Parallel Processing Environment.* PhD thesis, Arizona State University, July 1997.

[28] S. Sardesai, D. McLaughlin, and P. Dasgupta. Distributed Cactus Stacks: Runtime stack-sharing support for distributed parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[29] A. S. Tanenbaum and S. Mullender. An overview of the Amoeba distributed operating system. *Operating Systems Review*, 15(3):51–64, July 1981.

Biography

Arash Baratloo is a Ph.D. candidate at the Courant Institute of Mathematical Sciences, New York University. He has received M.Eng. and B.S. degrees in computer science from Cornell University. His current research interests include load balancing and fault tolerance in distributed and parallel systems.

Partha Dasgupta is an Associate Professor at Arizona State University. He received his Ph.D. in Computer Science in 1984 from SUNY Stony Brook. His research interests encompass distributed computing, middleware, operating systems and parallel computing infrastructure.

Vijay Karamcheti is an Assistant Professor of Computer Science in the Courant Institute of Mathematical Sciences at New York University. Vijay received his Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign in 1998. His current research focuses on computer architecture, and system software issues in parallel and distributed computing.

Zvi Kedem received his D.Sc. in Mathematics from the Technion - Israel Institute of Technology in 1974. He has published in: functional analysis, algebraic computational complexity, VLSI complexity, analysis and design of sequential and parallel algorithms, computer graphics, concurrency control, databases and knowledge-based systems, data mining, compilation for special-purpose processor arrays, distributed systems, and metacomputing. He is a fellow of the ACM.

An Overview of MSHN: The Management System for Heterogeneous Networks

Debra A. Hensgen[†], Taylor Kidd[†], David St. John[§], Matthew C. Schnaidt[†], Howard Jay Siegel[‡], Tracy D. Braun[‡], Muthucumaru Maheswaran[¥], Shoukat Ali[‡], Jong-Kook Kim[‡], Cynthia Irvine[†], Tim Levin[§], Richard F. Freund[μ], Matt Kussow[μ], Michael Godfrey[μ], Alpay Duman[†], Paul Carff[†], Shirley Kidd[§], Viktor Prasanna[¶], Prashanth Bhat[¶], and Ammar Alhusaini[¶]

[†]Department of Computer Science
Naval Postgraduate School
Monterey, CA USA

[‡]School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN USA

[¶]Electrical and Computer Engineering
University of Southern California
Los Angeles, CA USA

[μ]NOEMIX
San Diego, CA USA

[¥]Department of Computer Science
University of Manitoba
Winnipeg, Canada

[§]Anteon Corporation
Monterey, CA USA

Abstract

The Management System for Heterogeneous Networks (MSHN) is a resource management system for use in heterogeneous environments. This paper describes the goals of MSHN, its architecture, and both completed and ongoing research experiments. MSHN's main goal is to determine the best way to support the execution of many different applications, each with its own quality of service (QoS) requirements, in a distributed, heterogeneous environment. MSHN's architecture consists of seven distributed, potentially replicated components that communicate with one another using CORBA (Common Object Request Broker Architecture). MSHN's experimental investigations include: (1) the accurate, transparent determination of the end-to-end status of resources; (2) the identification of optimization criteria and how non-determinism and the granularity of models affect the performance of various scheduling heuristics that optimize those criteria; (3) the determination of how security should be incorporated between components as well as how to account for security as a QoS attribute; and (4) the identification of problems inherent in application and system characterization.

1. Introduction

The Management System for Heterogeneous Networks (MSHN[1]) project seeks to determine an effective design for a resource management system (RMS) that can deliver, whenever possible, the required quality of service (QoS) to individual processes that are contending for the same set of distributed, heterogeneous resources. Factors influencing QoS requirements include security, user preferences for different versions of an application, and deadlines. A set of QoS requirements, considered together with resource availability, determine whether all processes' requirements can be met.

An RMS, also sometimes called a meta-computing system, is similar to a distributed operating system in that it views the set of machines that it manages as a single virtual machine [50]. Also, like any distributed operating system, it attempts to give the user a location-transparent view of the virtual machine. Hence, as in the case of a distributed operating system, an RMS provides users with improved performance while the location of resources is hidden. The set of users of a system, which consists of both local and remote resources, that is managed by an RMS should be able to attain a higher level of availability and more fault tolerance than would be available from their local system alone.

This research was supported, in part, by the DARPA/ITO Quorum Program.

[1] Pronounced, "mission"

0-7695-0107-9/99 $10.00 © 1999 IEEE

An RMS differs from a distributed operating system in that it does not micro-manage the resources of each computer. Instead, each computer runs its native operating system. Similarly, each router executes its own protocol and each file server executes a native distributed file system. The RMS is responsible for identifying the large-grained resources, i.e., compute servers and data repositories that should be used by each process, if there is a choice. It may be responsible for issuing a command to begin execution of the processes that comprise an application. It may monitor the status of both the resources in the system and the progress of the applications for which it is responsible.

It is unclear whether every request to execute an application that is submitted to any operating system on any of the machines in the distributed system must be controlled by the RMS. If all requests are controlled by the RMS, then allocation policies that attempt to optimize throughput for a set of well-understood applications will perform better. However, sometimes users wish to maintain control over which resources their application will use.

There are many active, on-going research projects, in addition to MSHN, in the area of resource management, and there are many major research problems to be solved [38]. A problem that MSHN is not addressing is the best way for such a system to interact with human users to obtain their QoS preferences and requirements in the most user-friendly way. Indeed, simply identifying the syntax and semantics required to express all of the QoS preferences and requirements is a difficult problem [11][17][37][54]. While MSHN does not address this problem, the designers of MSHN expect to leverage results from research in this area. They assume, for example, that a request to execute an application is accompanied by a list of deadlines, preferences for various versions of an application, security requirements, and any restrictions on the variance of the time at which a request should be completed.

Before leaving the general topic of RMSs, it is imperative that we address the topic of "packaging." MSHN researchers do not see the fruits of the RMS research as a large, monolithic piece of software that will require its own separate installation and maintenance. The best way to package the eventual outcomes of the RMS projects may be to incorporate them into an infrastructure- or middleware-level standard similar to the Common Object Request Broker Architecture (CORBA), Domain Name Services, or other such resource location services. In this way, an RMS would not need to be separately maintained and would be consolidated with the services that distributed applications will most often use. However, it is still worthwhile to separate research on RMSs from research in all other aspects of distributed object computation that will be needed in future versions of such standards in order to first isolate, then solve some of the difficult resource management problems.

1.1. Background

MSHN evolved in part from a scheduling framework called SmartNet [19][28]. SmartNet's goal was to be able to wisely schedule sets of compute-intensive jobs, some of which may require the execution of multiple processes, onto members of a suite of heterogeneous computers. SmartNet provides a sophisticated scheduling module that had been successfully integrated with many RMSs and distributed computing environments. Hence, users who need to execute compute-intensive jobs and have access to a shared, heterogeneous environment can achieve superior performance, while continuing to work in an environment to which they have grown accustomed [21]. Additionally, for those users who do not already have one installed, SmartNet provided a basic RMS that makes use of its sophisticated scheduling capabilities. SmartNet's major research contributions include:

- The ability to predict the expected run-time of a job on a machine using the concept of compute characteristics and information collected from previous executions of the job.
- The ability to leverage the heterogeneity inherent in both a collection of jobs as well as in a collection of computers.

SmartNet was used successfully by DoD and the National Institutes of Health in scheduling their compute-intensive jobs, and by NASA's EOSDIS system in determining whether their resources were adequate to process data in the ways desired by their scientists.

SmartNet's scheduling algorithms are tuned to attempt to minimize the time at which the last job completes, although the designers of SmartNet recognized that similar algorithms may be useful in optimizing other criteria.Of course, minimizing the time at which the last job, of a set of jobs, completes is, in general, an NP-complete problem, so SmartNet employs heuristics when it searches for a near-optimal mapping of jobs to machines and job execution schedule. Many of the heuristics that it uses are well known and previously documented, however, they had not previously been used in a practical heterogeneous computing system [25]. It is likely that they were not previously used in actual systems because system designers had not tried to estimate average process run-times and because it was not previously recognized that exact run-times, though helpful, were not necessary [2][3][26].

1.2. Overview of MSHN's goals

MSHN differs from SmartNet in three major ways. First, SmartNet was expected, from the beginning, to be a

system that would actually be used in production. For this reason, much of the SmartNet developers' time was spent ensuring that SmartNet was at SEI Level 3. Despite this, SmartNet was able to make significant research contributions. MSHN is intended to be a research system, facilitating experiments by the investigators to determine how RMSs, that have somewhat broader goals than SmartNet, can be built. MSHN's research goals expanded upon SmartNet's in the following areas.

(i) MSHN needs to consider that the overhead of jobs sharing resources, such as networks and file servers, can have significant impact on mapping and scheduling decisions.

(ii) MSHN must support adaptive applications (defined below).

(iii) MSHN must deliver good QoS to many different sets of simultaneous users, some of whom may be executing interactive jobs; others, compute-intensive jobs; and still others, real-time requirements.

In SmartNet's model, applications consist of three distinct phases. In the first phase, which is short compared to the second phase, they acquire data from a data repository. In the second phase, they compute results based upon the data that they obtained during the first phase. In the third phase, which is again very short compared to the second phase, they write the result back to a possibly different repository. Because the first and third phases are so short, SmartNet's heuristics assume that there is no contention for either the network or the data repositories. However, they do account for the time required to access the resources, assuming that each application is the sole user of those resources. The model of applications that MSHN is meant to manage is more complex, permitting applications to transition through many more phases of variable length, each requiring not only sharing of compute resources, but also sharing of network and data repository resources. We discuss briefly in this paper, and elaborate elsewhere, both the problem of modeling the application and that of accounting for lower level policies that govern the sharing of resources. That is, because MSHN does not assume that it has any control over network routing, file server memory allocation, etc., it models, when necessary, the lower level operating systems and protocols. By doing so, the assignment of processes to resources will account for the sharing of those resources in the correct way.

The second major difference between SmartNet and MSHN's research goals is that MSHN attempts to provide support for **adaptive** and **adaptation-aware** applications. By adaptive applications, we mean idempotent applications that can exist in several different versions. Different versions may have different valuesto a user due to factors such as precision of computation or input data. Additionally, different versions may have different communication and computation needs. Or, one version may execute on Windows NT while another version is an executable for Linux. MSHN's goal is to support adaptive applications by being able to terminate one version of an application if MSHN perceives that the currently executing version will not meet the users' QoS expectations.[2] In that case, MSHN would terminate the executing version and start up another version from the beginning (if there were sufficient resources to execute that other version). The requirement that adaptive applications be idempotent permits the application to be safely restarted from the beginning without corrupting any resource such as a database. Similarly, there may be times when MSHN determines that delivery of a better QoS is possible to a user by changing to a version that better meets that user's preferences.

An adaptation-aware application differs from an adaptive application in two ways. First, when it is terminated, the new version need not be restarted from the beginning. Instead, a different version from the one that terminated may be started, using information about a previous state that was obtained from the execution of the previous version. Second, an adaptation-aware application may be able to adapt its resource usage during execution, without restarting.

Finally, MSHN's goals differ from SmartNet's in that MSHN seeks to determine how to meet multiple different QoS requirements to multiple different applications simultaneously. There are really two issues bound up in this difference. First, a way to incorporate, dynamically, the mixture of QoS requirements into a single measure must be determined. Second, an assignment of applications to resources must also be determined that optimizes the identified measure. In resolving this second issue, we can strongly leverage SmartNet's emphasis on the separation of optimization criteria and search algorithms and the recognition that similar algorithms can be used to search many different types of spaces for optimal values. We elaborate on this below.

1.3. Related work

There are other research groups examining the issues important to building an RMS, many within DARPA's Quorum project. Here, we look at some of the projects related to MSHN. Some of these groups are engaged in research complementary to MSHN's goals. For the sake of brevity, only a short synopsis of each project, as it relates to MSHN, is presented.

DeSiDeRaTa. The University of Texas at Arlington has a project called 'DeSiDeRaTa: QoS Management Tools for Dynamic, Scalable, Dependable, Real-Time

[2] We note that a version of one application may be terminated because MSHN detects that another user's application will not meet its QoS expectations. This phenomenon can occur due to priorities.

Systems." DeSiDeRaTa is focusing on QoS specification, QoS metrics, dynamic QoS management, and benchmarking of specific computing environments, such as the distributed Anti-Air-Warfare system at the Naval Surface Warfare Center, Dahgren Division. A unique concept that has come out of the DeSiDeRaTa project is that of an application "path" [56].

Globus. Globus is a large, joint project from Argonne National Laboratory and the University of Southern California's Information Sciences Institute. Parts of the Globus project are devoted toward resource management issues. The Globus architecture depends on an advance or immediate resource reservation protocol layer, for which a standard does not yet exist [14][18].

RT-ARM. Honeywell is developing a "Real-Time Adaptive Resource Management" system aimed primarily at high-end, real-time military embedded systems such as the Navy Surface Combatant Ship SC-21. Some of the specific issues they are concentrating on include modeling embedded systems and finding practical techniques for predictable real-time performance[24].

EPIQ. The EPIQ project, from the University of Illinois at Urbana-Champaign, is building an infrastructure for providing guaranteed QoS features, upon which RMSs may be built. part of their infrastructure involves building their own runtime environment[33].

ERDoS. SRI International is running a project called ERDoS (End to End Resource Management for Distributed Systems) which is developing an architecture for adaptive QoS-driven resource management. The ERDoS project emphasizes a comprehensive definition of QoS and the development of models that capture information required for making resource management decisions [46].

QUASAR. The QUASAR (QUAlity Specification and Adaptive Resource management for distributed systems) project, at the Oregon Graduate Institute of Science and Technology, is investigating techniques for specifying and utilizing QoS in adaptive, distributed systems. QUASAR is concentrating on the translation of QoS specifications from the application-level to the resource-management-level, and its use in reservation-based resource management, primarily in the multimedia domain[51].

ASSERT. The ASSERT System at the University of Oregon, Eugene, is focusing on dynamic, distributed, real-time environments. The core of the project estimates and monitors the relevant QoS parameters of running applications. ASSERT is not an RMS, nor an RMS framework; rather, the ASSERT project is looking at a specific issue of RMSs: QoS monitoring and estimation [15].

QuO. The Quality Objects (QuO) project, from BBN Systems Technologies, is attempting to add QoS specification and delivery to CORBA. Rather than provide absolute QoS guarantees, QuO seeks to combine knowledge about resource and application conditions in order to reserve enough end-to-end resources for predictable execution of distributed applications[47].

MOL. The MOL (Metacomputing OnLine) project from the Paderborn Center for Parallel Computing has as a goal the utilization of multiple high performance systems for solving problems too large for a single supercomputer. The MOL approach does not assume absolute control of resources under its management. The MOL project is addressing several of the issues key to resource management, including QoS specification[40].

1.4. Organization of the paper

In the next section of the paper we motivate and discuss MSHN's architecture. Even though SmartNet was successful in achieving its functionality, rather than using SmartNet's architecture exactly, we based MSHN's architecture upon lessons learned from SmartNet, because MSHN's goals are substantially different. In particular, we clearly delineated certain of SmartNet's modules into separate components. This delineation makes it easier to experiment with different designs for each of the components. In section 3, we then discuss many of the research issues that the MSHN investigators are studying and highlight some of the results. Additionally, this section provides references to the numerous articles that describe this research in more detail. We conclude by summarizing the status of the MSHN project.

2. MSHN's architecture

In this section, we first describe some of the concepts that went into MSHN's architectural design. This description motivates the need for the various major components and explains why they must be replicated to varying degrees. The architectural design was driven by the need to support the RMS research that we will discuss in the next section and was aided by our previous experience with SmartNet. We then present MSHN's current architecture in detail.

2.1. Motivation

We first motivate the need for each of the major components of MSHN's architecture, then discuss how those components interact with one another.

We recall from the previous section that an RMS needs to transparently locate the resources that should be used when execution of an application is requested. Therefore, it must be made aware of any request, by either a user or an application, to start executing another application. Many early RMSs required the user to explicitly log in to the system to start a job. If an application was to be started from within another application, e.g., through

`fork` and `exec` system calls, then the application that makes the request would be required to be specially designed to embed these requests within a function call to an RMS library. This restriction required that applications be specifically written or modified for a particular RMS.

The MSHN designers do not want to force a user to explicitly log into an RMS, or to modify their existing programs. Instead, MSHN transparently intercepts calls to system libraries that would otherwise initiate execution of a new process and diverts those calls to a MSHN Client Library. After MSHN decides where the newly requested application should execute, the MSHN Client Library uses whatever mechanisms available at the resource site to initiate execution of the remote process.

The environments for which MSHN is designed contain many different types of computers, each possibly executing a different version of an operating system. Rather than requiring the Client Library, which is linked with every MSHN application, to contain a substantial amount of code that is specific to each of these computers, we chose to make use of a MSHN Daemon. Whenever a computer is added to a system, a MSHN Daemon is started on that computer. When a Client Library needs to start a process on a remote machine, it simply contacts the MSHN Daemon on that machine and requests that the Daemon start the process on the Client Library's behalf. Of course, the general mechanism that we use in the Daemon is not new, and is therefore not a research issue.

When a remote process needs to communicate with the initiating process, it contacts the Client Library, which passes the information on to the initiating process, just as though the remote process were started locally. Being able to transparently provide this service to applications, whether or not they are command interpreters, requires that the Client Library intercept, and at least pre-process if not divert, other system library calls in addition to the previously mentioned `exec` call. For example, all of the socket calls and all calls to open, close, read, and write files must be intercepted and replaced or at least pre- and post-processed.

The MSHN project required a mechanism for intercepting these calls without requiring source modification. We initially turned to the Condor project for help with this problem [36]. Condor is a project at the University of Wisconsin that performs transparent migration of processes in a Unix environment. To perform this migration, Condor also had to intercept these calls to system libraries. Using techniques similar to those used by Condor, we were able to intercept these calls without requiring source code modification.[3] The mechanism is described in detail elsewhere[43].

In addition to providing a mechanism for transparently executing remote processes, the Client Library is in a unique position to passively determine the status of resources, because it is assumed to be linked with any application executing in an environment managed by MSHN. That is, the MSHN Client Library can pre- and post-process system calls, because it is intercepting all such calls made to the operating system, which are executed when a process needs to use a hardware resource. In so doing, it can determine the low level, end-to-end QoS that an application is receiving from a particular resource. We will discuss this functionality of the Client Library further in the next section.

When the MSHN Client Library intercepts a call to execute a new process, it must have some way of determining which resources that new process should use, i.e., which computer should primarily be responsible for executing the new process.[4] Rather than requiring that decision to be made independently by each Client Library that is linked with each application, we chose to have the Client Library first check the request against a list of applications managed by MSHN. If the requested application is not on that list, the MSHN Client Library simply passes the requested application directly to the local operating system. If the requested application is on that list, it instead passes the request to the MSHN Scheduling Advisor. It is the Scheduling Advisor's job to determine which set of resources the newly requested process should use.

The MSHN Scheduling Advisor is itself a complex package, associated with many different research issues which we discuss more fully in the next section. Among the primary research issues are: (i) what criteria should be optimized in the choice of resources? (ii) Because optimizing the criteria is likely to be an NP-complete problem, if n is too large, which heuristic should be used to search for an optimum resource assignment? (iii) With what granularity must the Scheduling Advisor model both the policies and protocols associated with allocation of the lower level resources and what granularity of model should it use to define the resource requirements of a process?

For the Scheduling Advisor to determine a good assignment of resources for a process, it must know both which resources and how much of each resource would be required for a process to execute and meet its QoS requirements and preferences. Therefore, to assist the Scheduling Advisor in making its decision as to the

[3] These techniques, however, require that the object code files be linked with the MSHN Client Library, therefore they require object code files. However, another tool, the Executable Editing Library (EEL) which evolved from the University of Wisconsin's Paradyn project could be used to link an executable with the MSHN Client Library, instead[31].

[4] In modern systems, the choice of computer that is responsible for executing a process often carries with it, implicitly, a choice of file servers and other distributed resources such as networks. Therefore, when we say that MSHN chooses a computer to be responsible for executing a process, the choice of other resources external to that computer may be implicit in that assignment.

assignment of resources, we designed both the MSHN Resource Requirements Database and the MSHN Resource Status Server.

The Resource Status Server is a quickly changing repository that maintains information concerning the current availability of resources. Information is stored in the Resource Status Server as a result of updates from both the MSHN Client Library and the MSHN Scheduling Advisor. The Client Library can update the Resource Status Server as to the currently perceived status of resources, which takes into account resource loads due to processes other than those managed by MSHN. The Scheduling Advisor can provide expected future resource status based upon the resources that it expects will be used by the applications that it assigns. Additionally, the Resource Status Server can statistically process its historic knowledge to make predictions of resource status even further in the future.

As compared to the Resource Status Server, the information maintained by the MSHN Resource Requirements Database changes much more slowly. The Resource Requirements Database is responsible for maintaining information about the resources that are required to execute a particular application. Although the initial MSHN prototype only implements a single source for the information stored in this database (statistically analyzed historical information), we envision that many other on-going research projects will also serve as sources for this information.

MSHN's current source for the information that is maintained by the Resource Requirements Database comes from data collected by the MSHN Client Library when the application was previously executed. Although patterned after SmartNet in this way, and leveraging the concept of compute characteristics that SmartNet pioneered, MSHN does not collect the same information as SmartNet collects. SmartNet's information is coarse-grained; that is, it maintains only the total amount of wall-clock time that is required to execute a program from beginning to end for each particular machine. This measure is sufficient for SmartNet's needs due to the requirements of its intended applications (three phases) and the expected environment (each job has exclusive access to the resources that it is using). However, in MSHN, resources are shared and applications have more phases, so maintaining only this coarse grain information is insufficient. Therefore, the Resource Requirements Database has the ability to maintain very fine grain information collected by the MSHN Client Library. Eventually it is hoped that the Resource Requirements Database can also be populated with information from smart compilers and possibly advice from application writers.

Applications, of course, are needed to test any system. Unfortunately, executables for many different platforms would be needed to test MSHN's ability to manage them in a distributed, heterogeneous environment. Producing such actual applications would require tremendous effort to obtain the source code for numerous applications, some of which may be classified or proprietary, port the source code to the different platforms, and compile and link them. We decided that this effort was better spent on our research system itself, so we looked for another viable solution. One solution that we considered was to use benchmarks, because many of them have already been ported to many different platforms. However, we wanted to make sure that our system could manage a wide variety of applications. We finally settled on writing a general-purpose application emulator whose parameters could be specified to cause it to imitate a wide variety of applications. We discuss the problem of deciding how best to construct such an emulator under the research topics in the next section.

The Client Library, which is linked with each executing MSHN application, informs the Resource Status Server about the current perceived status of the resources that the applications are using. The Scheduling Advisor informs the Resource Status Server only about the load that it expects the processes, which it has scheduled, to place on certain resources. However, neither class of information indicates the condition of resources that no MSHN application is currently using or is planning on using. Therefore, we use a MSHN Application Emulator linked with the Client Library to obtain information about the condition of such resources.

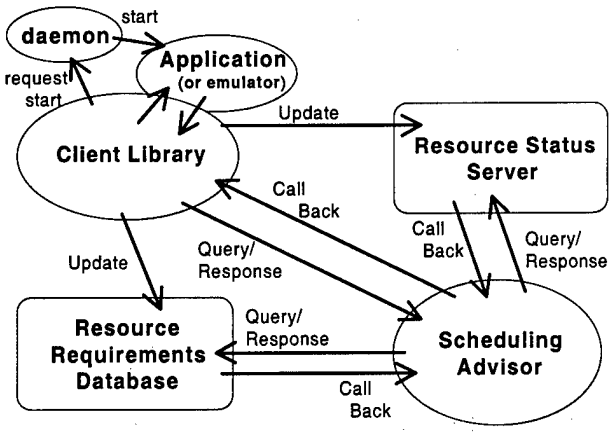


Figure 1 MSHN's conceptual architecture.

MSHN's conceptual architecture is shown in Figure 1. As can be seen in the figure, every application running with MSHN makes use of the MSHN Client Library that intercepts the application's operating system calls. When the Client Library intercepts a request to execute a new application, and that application requires that the MSHN Scheduling Advisor be consulted to determine the resources that the application should use, the Client

Library invokes a scheduling request on the Scheduling Advisor. The Scheduling Advisor queries both the Resource Requirements Database and the Resource Status Server. It uses information that it receives from them, along with an appropriate search heuristic, to determine where the newly requested process should execute. After determining which resources should host the new process, the Scheduling Advisor returns the decision to the Client Library, which, in turn, requests execution of that process through the appropriate MSHN Daemon. The MSHN Daemon invokes the application on its machine. As a process executes, the Client Library updates both the Resource Status Server and the Resource Requirements Database with the current status of the resources and the requirements of the process. Meanwhile, the Scheduling Advisor establishes callbacks with both the Resource Requirements Database and the Resource Status Server. Using callbacks, the Scheduling Advisor is notified in the event that either the status of the resources has significantly changed, or the actual resource requirements are substantially different from what was initially returned from the Resource Requirements Database. In either case, if it no longer appears that the assigned resources can deliver the required QoS, the application must be adapted or terminated. Upon receipt of a callback, the Scheduling Advisor might require that several of the applications adapt so that more of them can receive their requested or desired QoS.

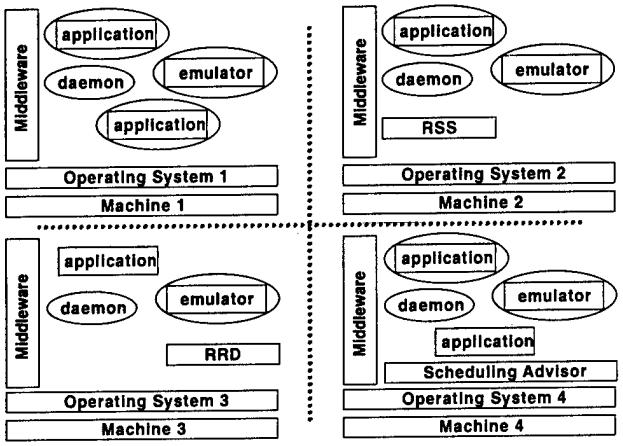


Figure2 Physical instantiation of the MSHN architecture.

Although all MSHN components could run on the same machine, they can also be distributed and replicated across many different computers using tools such as ISIS, Horus and Ensemble [7][49][48]. Results from control theory will also be useful here in ensuring that the process of replicating and merging components is stable and does not result in oscillation. Additionally, results from control theory must be incorporated into the replicated Scheduling Advisor itself to ensure that modifications requested of adaptive and adaptation-aware applications do not become unstable. MSHN components might even replicate as needed [20][21]. **Figure 2** illustrates a simple instantiation of the MSHN system.

In addition to the components discussed above, we found it convenient to add a MSHN Visualizer that enabled us to examine, for both functional and performance debugging purposes, the current states of the various MSHN components. The MSHN Visualizer captures all significant events within and between the core MSHN components for real-time and post-mortem analysis.

Security within the MSHN architecture has been considered. Policies of interest are:

- Component authentication. This includes authentication of MSHN core components to each other; authentication of resource-based clients to the MSHN core; and authentication of applications to selected MSHN components.
- Hierarchical least privilege. Within the MSHN context, the core components are the most privileged, while user applications are the least privileged.
- Communications integrity and confidentiality. Communications are protected from unauthorized modification and disclosure.
- Access control. Access to MSHN core databases and to job histories may be mediated.

The security architecture creates keyed domains, supporting least privilege, authentication, confidentiality and integrity by using the Common Data Security Architecture facilities for security services and key management[5] [57][58].

2.2.1. The current MSHN architecture. A high level description of the current MSHN architecture is presented. For a more detailed description, we refer the reader to other publications [43]. High-level diagrams are presented for each MSHN component, with arrows indicating the direction of communication or action. In addition to these diagrams, a short description of each component's functions is given. In the description of the MSHN architecture, we represent MSHN components and external components as Unified Modeling Language (UML) actors [8]. The symbols used for this representation are shown in Figure 3. The core MSHN components include the Scheduling Advisor (SA), the Client Library (CL), the Resource Status Server (RSS), the Resource Requirements Database (RRD), the Daemon (D) and the Application Emulator (AE).

[5] As in any RMS, assurance of MSHN's security properties is built on and limited by the effectiveness of the security environment provided by the underlying operating system(s) and hardware base(s).

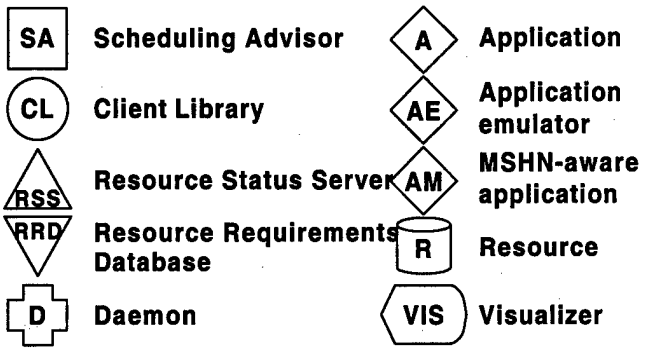
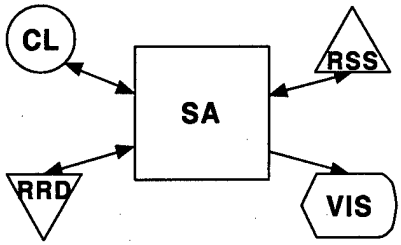
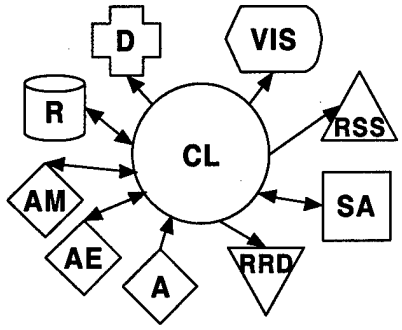


Figure3 Symbols representing actors in the MSHN architecture.

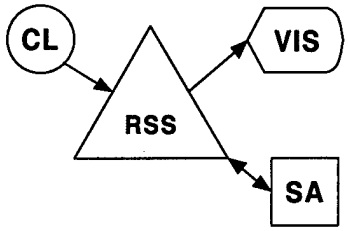
Scheduling Advisor (SA) functionality. The primary responsibility of the SA is to determine the best assignment of resources to a set of applications, based on the optimization of a global measure, which we describe in the next section. The SA depends on the RRD and the RSS in order to identify an operating point that optimizes the global measure. It responds to resource assignment requests from the CL. When appropriate, the SA requests application adaptations via the CL. The SA is also responsible for establishing callback criteria (thresholds) with the RSS and RRD. All MSHN components update the MSHN Visualizer with all significant display and post-mortem analysis events.



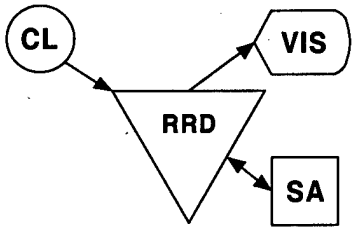
Client Library (CL) functionality. The CL is linked with both adaptive and adaptation-aware applications. It provides a transparent interface to all of the other MSHN components. The CL intercepts system calls to collect resource usage and status information, which is forwarded to the RRD and the RSS. The CL also intercepts calls that initiate new processes (such as `exec()`) and consults the SA for the best place to start that process. It requests (possibly remote) daemons to execute applications based on the SA's advice. The CL invokes adaptation on adaptation-aware applications when notified by the SA via callbacks. One such invocation is the special case of setting emulator parameters.



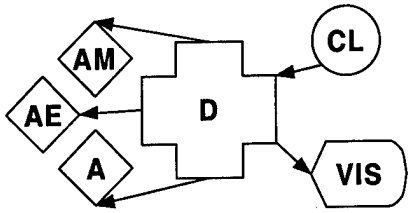
Resource Status Server (RSS) functionality. The role of the RSS is to maintain a repository of the three types of information about the resources available to MSHN: relatively static (long-term), moderately dynamic (medium-term), and highly dynamic (long-term) information. The RSS is updated with current data via the CL or through a system administrator. The RSS responds to SA requests with estimates of currently available resources. The SA sets up callbacks with the RSS based on resource availability thresholds and CL update frequency requirements.



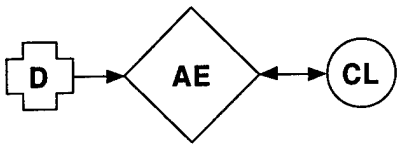
Resource Requirements Database (RRD) functionality. The RRD is a repository of information pertaining to the resource usage of applications. The RRD provides this information to the SA. Callbacks to the SA are based on either the occurrence of a threshold violation or update frequency requirements. It is updated by the CL.



Daemon (D) functionality. The MSHN Daemon executes on all compute resources available for use by the SA. Its sole purpose is to start applications as requested by the CL. It therefore has the capability and responsibility of initiating the default application emulator at start-up to determine resource status information.



Application Emulator (AE) functionality. The AE emulates a running application by stressing particular resources in the same way as the real application does. The AE serves two purposes: The first is to run simulated applications (that statistically leave the same resource usage footprint of the real applications) without the overhead and uncertainty of actually installing, maintaining, and running that particular application. The second is to be a monitor, in the absence of any other MSHN-scheduled applications. That is, it can determine the status of resources that are not being otherwise used by MSHN-scheduled applications, and therefore not being monitored by an existing CL. The Daemon starts one instance of the AE, by default, at startup. Other instances may be started at any other time through a command interpreter or other application.



3. MSHN Research Issues

In this section of the paper, we describe some of the major issues being investigated by the MSHN team members. We also briefly summarize some of the results to date. Of course, there is not sufficient space to completely describe all of the issues and results in detail, so the reader is also referred to relevant papers on each topic. We have attempted to associate the issues with the component of the MSHN architecture that they most strongly affect. However, certainly many issues that affect the Scheduling Advisor also affect the Resource Status Server and Resource Requirements Database. Additionally, this work is non-orthogonal to research being done by many investigators outside of the MSHN team who are examining such issues as how QoS requirements are derived from smart compilers and how they can be best expressed.

3.1. Scheduling Advisor research issues

In this section we discuss some issues that most strongly affect the Scheduling Advisor. First, we examine how to quantify the needs of all of the processes that require resource allocation by the Scheduling Advisor. Then, we consider the ramifications of not precisely knowing the resource requirements, and consequently, the exact future status of all of the resources. Finally, we discuss the class of heuristics that have thus far been implemented in MSHN and why there is a need for a variety of heuristics.

3.1.1. Optimization criteria. Optimal resource allocation always involves attempting to solve an optimization problem, which is usually NP-complete. SmartNet's primary optimization criterion was to minimize the time at which an application completes, assuming that all of the applications were of a particular form. Later versions of SmartNet also accounted for priorities. MSHN maximizes a weighted sum of values that represents the benefits and costs of delivering the required and desired QoS (including security, priorities, and preferences for versions), within the specified deadlines, if any. We now discuss the effect of each of these attributes on the optimization criteria.

- MSHN's consideration of security as an optimization criterion allows the trade-off of security with other QoS constraints when there are insufficient resources to complete all requests. This is done in a fashion similar to other recent projects [45]. MSHN associates a cost to security levels that varies, depending upon which resources are being used to obtain a given level of security (for more details on security viewed as a QoS parameter, see section 3.2).
- MSHN attempts to account for both preferences for various versions and priorities. That is, when it is impossible to deliver all of the most preferred information within the specified deadlines due to insufficient resources, MSHN's optimization criteria are designed to favor delivering the most preferred version to the highest priority applications.
- In MSHN's optimization criteria, deadlines can be simple or complex. That is, sometimes a user will be satisfied if a result is received before a specific time. Other times, a user would like to associate a more general benefit function, which would indicate that information might have different values based upon when it is received.

Further information about MSHN's optimization criteria can be found elsewhere [23][30].

In addition to a cost function that is optimized, optimization problems usually have a set of constraints that must be met in order for a solution to be viable. The constraints of a resource allocation optimization problem are that the resources allocated to meet the needs of the processes must be less than or equal to the available resources at any point in time. The actual inequalities required not only depend upon the QoS constraints, but

also upon the sharing policies used by the local operating systems and network protocols, and upon the granularity with which both those policies and resource usage should be known (see Granularity Issues in Section 3.2).

3.1.2. Inexact knowledge of job resource usage. Even if it is possible to find a perfect solution to the optimization problem that is posed by instantiating the constraints and optimization criteria to the current situation, the expected resource usage of any given application is often only an estimate. In real-time systems, the worst case estimate is often used to assign resources to processes; however, many other systems use the mean expected resource usage. Our recent analysis has revealed that using the mean will cause the actual run-time to be generally underestimated and that a better assignment can be made if both the mean and distribution of the expected resource usage is accounted for, when appropriate [26].

This leads to another question concerning whether the extra complexity involved in using a sophisticated heuristic will yield a better schedule than using a simple heuristic if the actual variance of run-times is large, and scheduling is done using the mean, or both the mean and the distribution. Our recent results in this area have shown that, in many cases, complex heuristics can determine schedules that, when executed, sometimes perform much better than the schedules derived from very simple heuristics, even when the variance is large. However, sometimes very simple heuristics perform just as well as the more complex ones. The difference in quality of the schedules produced by the various heuristics was found to be closely correlated with the type of heterogeneity in a system. For example, when both the machine and application heterogeneity is very low, a simple heuristic performs just as well as more complex ones. Several papers have described our results concerning this research [2][3][10][40].

3.1.3. Performance of search algorithms. SmartNet's organization leveraged the idea of independence of search algorithms and optimization criteria. That is, most heuristics for searching the space of mappings can be modified to search for solutions to different optimizations within the same space. For example, Dantzig's Simplex Method is useful with all problems whose optimization criteria and constraint inequalities can be stated using only linear combinations of the variables. Sometimes, many different heuristics will work, but, depending upon the characteristics of a given problem, certain heuristics may be preferable to others. For example, the MSHN team has obtained extensive results identifying the regions of heterogeneity where certain heuristics perform better than others for maximizing throughput by minimizing the time at which the last application, of a set of applications, should complete [2][3][10][40]. Re-targeting of these heuristics to other optimization criteria is currently underway.

Additionally, MSHN team members have performed extensive research into accounting for dependencies between applications or processes that make up a single application [40][51][52][54]. This includes promising results from investigating data dependencies and mapping of iterative applications [1][4][5][6][11].

3.2. Resource Status Server and Resource Requirements Database research issues

Part of the MSHN team's investigation has been aimed at determining what information should be stored in the Resource Requirements Database and maintained by the Resource Status Server. First, a taxonomy for the types of information that could be stored there was required. We discuss this taxonomy below. We also discuss the impact that viewing security as a QoS has on these two MSHN components. Finally, one of the most important issues in designing effective RMSs is determining the level of granularity of information that must be maintained concerning the status of resources and the requirements of applications. We now discuss each of these issues in somewhat more detail and refer the interested reader to relevant publications.

3.2.1. A taxonomy. The MSHN team has formulated a three-part taxonomy for classifying systems. The three different components include methods for describing the applications, the computing environment, and the mapping strategy that is used. Some of the relevant characteristics that need to be instantiated concerning each application include

(i) Its size, that is the number of tasks or sub-tasks associated with it.

(ii) Whether the sub-tasks are independent of one another or, if they are dependent, the types of dependencies.

(iii) The I/O distributions of the application and the sources of the I/O, i.e., whether it performs all input in the beginning and all output at the end or whether one or the other is performed continually throughout the lifetime of the processes and whether the input data is obtained through interacting with a person or some other source that has highly variable response times.

(iv) The deadlines and other QoS requirements, including security, if any, associated with the applications and/or the subtasks that comprise the application.

Similarly, the computing environments and mapping strategies have numerous, hierarchically characterizable, attributes that are more fully documented in other publications [9].

3.2.2. Security as a quality of service. Security in the context of QoS is a current research area [34][45]. The security capabilities of resources and security requirements of applications must influence the assignment of applications to resources. We can obtain information concerning the user security requirements from the Resource Requirements Database and information concerning the security capabilities of the resource from the Resource Status Server. For example, if the output of an application must be encrypted using a particular algorithm, with a key size chosen within a particular range, then that requirement must be stored in the Resource Requirements Database along with the amount of data that must be encrypted. Also, the Resource Status Server must know whether each particular computing resource is capable of performing the required cryptographic algorithm and the cost, in terms of run-time per byte, for example, of encrypting the data. Members of the MSHN team have developed an initial framework, which they are currently refining, for characterizing the overall security attributes of a network and for determining a cost and benefit value for providing required and preferred security to an application [26][27][33][33].

3.2.3. Granularity issues. Another very important question that concerns both the Resource Requirements Database and the Resource Status Server has to do with how much detail should be maintained concerning the status of resources and the requirements of applications. Obviously, while a very accurate, detailed set of information might prove quite useful to the scheduling algorithms, it would be at the least very expensive and difficult to collect if not expensive to process within the algorithm itself.

The MSHN team has obtained initial estimates for the overhead of capturing system calls to determine the cost of collecting various granularities of such information[43]. Members of the team are currently using this technique to record fine-grained information for a program that analyzes air tasking orders and will report both the information concerning the resources that were used, as well as the overhead involved in collecting the resource usage information [40].

In addition to the cost associated with collecting fine-grained information concerning applications' use of resources, there is the question of how much information is sufficient. Current experiments of the MSHN team focus on determining whether fairly simple models can be used to predict the relative performance of application/resource assignments. To perform realistic experiments, the team has built an initial application emulator (see below) and is actually executing it with different parameters on different systems, using all possible configurations to compare the actual received QoS to the predicted QoS. Thus far we have determined that the Resource Status Server must, directly or indirectly, contain information concerning whether native threads are supported by the operating system. If this information is not maintained, the scheduling algorithm, which must choose between two platforms that are identical except for the operating system version that they execute, may assign a process which could be handled better by one platform to the other. Similarly, the Resource Requirements Database must indicate whether or not the application is multi-threaded and the number and nature of threads that it uses. Information concerning these results can be found in other publications[11].

3.3. Application Emulator research issues

The MSHN team is designing and implementing an application emulator for two different reasons. One reason is that it is needed within the MSHN architecture to monitor the end-to-end status of the resources. The other reason is to be able to easily construct a very large suite of application emulators that place loads on resources in the same way that the actual applications would. When used in conjunction with resource usage measurements from linking actual applications to MSHN's Client Library, the MSHN Application Emulator can be used to emulate the execution of the actual applications without requiring the applications to actually be ported to many different platforms. The obvious advantage of using such an application emulator, rather than porting the applications themselves, is to enable the MSHN researchers to test their architecture more quickly under many different situations.

To meet the first purpose of the MSHN Application Emulator, we first had to define the meaning of loading resources for various resources. Percentages cannot be used, as they are not transferable between either computing platforms or network media. Rather, each category of resource was identified and units that can be most easily translated between different platforms, such as FLOPS and bytes/sec, were chosen to quantify resource use. Also recognized at this stage was the need to have both multi-threaded and non-multi-threaded application emulator capability. Finally, not only can a single application be comprised of multiple threads, but it can also be comprised of multipleheavy-weight processes.

When designing the Application Emulator to meet both of its requirements, we recognized that distributions reflecting communication and computation alone were insufficient; conditional probabilities were required. That is, many times the purpose of one process sending a message to another process is so that the receiving process will perform work on behalf of the sending process. Therefore, we designed our most general emulator to also have the capability of sending work-bearing messages.

To this end, we have completed an initial implementation of an application emulator that we have used for our granularity research and are testing the more general application emulator. Documentation concerning both of these application emulators can be found elsewhere [11][15].

3.4. Client Library research issues

The research issues having to do with the Client Library component involve both mechanism and policy. The mechanism issues have to do with how to transparently link the Client Library with applications. Previous research in the areas of process migration and tools for debugging parallel and distributed programs provide us with easy solutions, as mentioned earlier. Therefore, the only issue that remains is how best to transparently determine the end-to-end availability of resources. First, simply determining that the Client Library could perform this functionality better than providing the functionality external to the applications themselves is an important contribution. However, determining the average end-to-end availability of a network resource is not a trivial problem. The MSHN team's initial progress in this area has already been detailed elsewhere [43].

4. Summary and future work

In this paper we summarized the purpose of a resource management system (RMS) in general and the research goals of one particular experimental RMS, the Management System for Heterogeneous Networks (MSHN). Motivation was provided for all of the major components of MSHN, and the architecture that contains those components was explained. Some of the research questions that the MSHN researchers are seeking answers to were described. References were provided that enable the reader to better understand MSHN, and to learn more about the MSHN experiments. There are many other interesting RMS research projects in progress today, but space permitted us to survey only a few of them. In addition to continuing the on-going experiments described in the paper, future MSHN investigation will focus on (i) reaching a better understanding of the level of granularity obtainable from applications and the level required to perform sufficiently good resource assignment; (ii) more detailed characterization of security costing and metrics; and (iii) determining the best search algorithms to use for the MSHN optimization criteria under various conditions.

Acknowledgments – The authors thank Shushanna St. John and LCDR Wayne Porter for their comments.

References

[1] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, "A unified resource scheduling framework for heterogeneous computing environments," *Proc. 8th IEEE Heterogeneous Computing Workshop*, April 1999.

[2] R. Armstrong, *Investigation of Effect of Different Run-Time Distributions on SmartNet Performance*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Sept. 1997.

[3] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions" *Proc. 7th IEEE Heterogeneous Computing Workshop (HCW 98)*, March 1998, pp. 79-87.

[4] P. B. Bhat, V. K. Prasanna, and C.S. Raghavendra, "Adaptive communication algorithms for distributed heterogeneous systems," *Proc. IEEE Intl. Symp. High Performance Distributed Computing*, July 1998, pp. 310-321.

[5] P. B. Bhat, V. K. Prasanna, and C.S. Raghavendra, "Block-cyclic redistribution over heterogeneous networks," *Proc. ISCA Intl. Conf. Parallel and Distributed Computing Systems*, Sept. 1998, pp. 242-249.

[6] P. B. Bhat, V. K. Prasanna, and C.S. Raghavendra, "Efficient collective communication in distributed heterogeneous systems," *Proc. IEEE Intl. Conf. Distributed Computing Systems*, 1999, to appear.

[7] K. Birman, "Replication and fault-tolerance in the ISIS system," *10th ACM Symposium on Operating Systems Principles*, Dec. 1985, pp. 79-86.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, Reading, MA, 1999.

[9] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems," *Proc. IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1998, pp. 330-335 (included in the proceedings of the *7th IEEE Symposium on Reliable Distributed Systems*, 1998).

[10] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," *Proc. 8th IEEE Heterogeneous Computing Workshop*, April 1999, to appear.

[11] J. R. Budenske, R. S. Ramanujan, and H. J. Siegel, "A method for the on-line use of off-line derivedremappings of iterative automatic target recognition tasks onto a particular class of heterogeneous parallel platforms," *The Journal of Supercomputing*, Vol. 12, No. 4, Oct. 1998, pp. 387-406.

[12] P. Carff, *Granularity*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, March 1999.

[13] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Resource Management for Value-Added Customizable Network

Service," *Proc. 6th IEEE International Conference on Network Protocols*, October 1998.

[14] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[15] T. Drake, *A Load Emulator Toolkit and Analysis of HiPer-D Resource Requirements*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, June 1999.

[16] S. Fickas, and M. S. Feather, "Requirements Monitoring in Dynamic Environments," *Proc. 2nd IEEE Intl. Symposium on Requirements Engineerings*, March 1995.

[17] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, "A Directory Service for Configuring High-Performance Distributed Computations," *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, 1997, pp. 365-375.

[18] I. Foster, and C. Kesselman, "The Globus project: a status report, " *Proc. 7th IEEE Heterogreneous Computing Workshop*, 1998, pp. 4-18.

[19] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Kieth, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet," *Proc. 7th IEEE Heterogeneous Computing Workshop*, March 1998, pp. 184-199.

[20] D. Hensgen, *Squads: Server Groups that Dynamically Adapt to Improve Performance*, Ph. D. Dissertation, Department of Computer Science, University of Kentucky, 1989.

[21] D. Hensgen, and R. Finkel, "Dynamic server squads in Yackos," *Proc. Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Oct. 1989.

[22] D. Hensgen, L. Moore, T. Kidd, R. F. Freund, E. Keith, M. Kussow, J. Lima, and M. Campbell, "Adding rescheduling to and integrating Condor with SmartNet," *Proc. 4th IEEE Heterogeneous Computing Workshop*, April 1995, pp. 4-11.

[23] D. Hensgen, T. Kidd, H. J. Siegel, J. K. Kim, D. St. John, C. Irvine, T. Levin, V. Prasanna, and R. Freund, *A performance measure for distributed heterogeneous networks based on priorities, deadlines, versions, and security*, Technical Report, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, Feb. 1999.

[24] J. Huang, R. Jha, W. Heimerdinger, M. Muhammad, S. Lauzac, B. Kannikeswaran, K. Schwan, W. Zhao and R. Bettati, "RT-ARM: a real-time adaptive resource management system for distributed mission-critical applications", *Workshop on Middleware for Distributed Real-Time Systems*, 1997.

[25] O. Ibarra and Kim, "Heuristic algorithms for scheduling independent tasks on non-identical processors," *Journal of the ACM*, Vol. 24, No. 2, 1977, pp. 280-289.

[26] C. Irvine and T. Levin, *A note on mapping user-oriented security policies to complex mechanisms and services*, Technical Report, Department of Computer Science, Naval Postgraduate School, Monterey, CA, in progress.

[27] C. Irvine and T. Levin, *Toward a taxonomy and costing method for security metrics*, Technical Report, Department of Computer Science, Naval Postgraduate School, Monterey, CA, in progress.

[28] T. Kidd and D. Hensgen, *Why the Mean is Inadequate for Making Scheduling Decisions*, Technical Report, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Jan. 1999.

[29] T. Kidd, D. Hensgen, R. Freund, and L. Moore, "SmartNet: a scheduling framework for heterogeneous computing," *Proc. 2nd Intl. Symposium on Parallel Architectures, Algorithms, and Networks*, June 1996, pp. 514-521.

[30] J. P. Kresho, *Quality Network Load Information Improves Performance of Adaptive Applications*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Sept. 1997.

[31] J. P. Kresho, D. Hensgen, T. Kidd, and G. Xie, "Determining the accuracy required in resource load prediction to successfully support application agility," *Proc. 2nd IASTED Intl. Conf. European Parallel and Distributed Systems*, July 1998, pp. 224-254.

[32] J. Larus and E. Schnarr, "EEL: machine-independent executable editing," *SIGPLAN PLDI 95*, 1995.

[33] T. Levin and C. Irvine, *An approach to characterizing resource usage and user preferences in benefit functions*, Technical Report, Department of Computer Science, Naval Postgraduate School, Monterey, CA, in progress.

[34] T. Levin and C. Irvine, *Quality of security service in a resource management system benefit function*, Technical Report, Department of Computer Science, Naval Postgraduate School, Monterey, CA, in progress.

[35] J. W. S. Liu, K. Nahrstedt, D. Hull, S. Chen, and B. Li, *EPIQ QoS Characterization, Draft Version*, July 1997.

[36] M. Livny, M. Litzkow, T. Tannenbaum, and J. Basney, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," *Dr Dobbs Journal*, Feb. 1995.

[37] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, "Specifying and measuring quality of service in distributed object systems," *Proc. 1st Intl. Symposium on Object-Oriented Real-Time Distributed Computing*, April 1998, pp. 20-22.

[38] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," *Proc. 8th IEEE Heterogeneous Computing Workshop*, April 1999, to appear.

[39] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous distributed computing," *Encyclopedia of Electrical and Electronics Engineering*, J. Webster, ed., John Wiley & Sons, New York, NY, to appear 1999.

[40] M. Maheswaran and H. J. Siegel, "A dynamic matching and scheduling algorithm for heterogeneous computing systems," *Proc. 7th IEEE Heterogeneous Computing Workshop*, Mar. 1998, pp. 57-69.

[41] N. W. Porter, *Resource Requirement Analysis of GCCS Modules and EADSim and Determination of Future Adaptivity Requirements*, Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, June 1999.

[42] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, J. Simon, T. Römke, and F. Ramme, "The MOL

project: an open extensible metacomputer," *Proc. 6th IEEE Heterogenous Computing Workshop*, April 1997.
[43] D. St. John, S. Kidd, D. Hensgen, T. Kidd, and M. Shing, *Experiences using semi-formal methods in MSHN*, Technical Report, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Feb. 1999.
[44] M. C. L. Schnaidt, *Design, Implementation, and Testing of MSHN's Application resource Monitoring Library* , Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, Dec.1998.
[45] P. Schneck and K. Schwan. "Dynamic authentication for high-performance networked applications," *Proc. 6th IEEE/IFIP Intl. Workshop on Quality of Service*, May 1998.
[46] J. Sydir, B. Sabata, and S. Chatterjee, " QoS middleware for the next-generation Internet," position paper, *Proc. NASA/NREN Quality of Service Workshop*, Aug. 1998.
[47] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken. "QuO's runtime support for quality of service in distributed objects," *Proc. IFIP Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing* , Sept. 1998.
[48] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, *Building adaptive systems using Ensemble*, Cornell University Technical Report, TR97-1638, July 1997.
[49] R. van Renesse, K. Birman, and S. Maffeis, "Horus, a flexible group communication system," *Communications of the ACM*, April 1996.
[50] R. van Renesse and A. S. Tanenbaum, "Distributed operating systems," *ACM Computing Surveys*, Vol. 17, No. 4, Dec. 1985.
[51] M. Tan and H. J. Siegel, "A stochastic model for heterogeneous computing and its application in data relocation scheme development," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 11, Nov. 1998.
[52] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li, "Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 8, Aug. 1999, pp. 857-871.
[53] J. Walpole, C. Krasic, L. Liu, D. Maier, C. Pu, D. McNamee, and D. Steere, "Quality of service semantics for multimedia database systems," *Proc. Data Semantics 8: Semantic Issues in Multimedia Systems IFIP TC-2 Working Conference*, Jan. 1999.
[54] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, Vol. 47, No. 1, Nov. 199, pp. 8-22.
[55] L. R. Welch, B. Ravindran, B. A. Shirazi, and C. Bruggeman, *Specification and modeling of dynamic, distributed real-time systems*, Technical Report Number TR-CSE-98-003, Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX, Sept. 1998.
[56] L. R. Welch, B. Ravindran, B. A. Shirazi, and C. Bruggeman, "DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable, Real-time Systems," *Proc. 15TH IFAC Workshop on Distribute Computer Control Systems*, Sept. 1998.
[57] R. E. Wright, *Management System for Heterogeneous Networks Security Services*, Thesis, C4I Academic Group, Naval Postgraduate School, Monterey, CA, June 1998.
[58] R. E. Wright, D. J. Shifflett, and C. E. Irvine, "Security for a virtual heterogeneous machine," *Proc. 14th Computer Security Applications Conference*, Dec. 1998, pp. 167-177.

Biographies

Debra Hensgen received her PhD in the area of Distributed Operating Systems from the University of Kentucky. She is an Associate Professor in the CS Department at The Naval Postgraduate School. She has co-authored numerous papers about the Concurra toolkit for automatically generating safe, efficient concurrent code, the Graze parallel processing performance debugger, the SAAM path information base, and the SmartNet and MSHN Resource Management Systems.

Taylor Kidd obtained his PhD from the University of California at San Diego in 1991. His interests include both theoretical and applied distributed computing. He led the research component of the SmartNet team at NRaD and instigated a number of important advances to the SmartNet Scheduling Framework. He is a co-PI for the DARPA-sponsored MSHN project and a co-investigator on the DARPA-sponsored SAAM Project.

David St. John is the head of staff at the Heterogeneous Network & Computing Laboratory, Naval Postgraduate School. He has over six years experience in object-oriented software development in the areas of distributed computing, process control, sensor collection, and Internet transaction processing systems. He is a member of IEEE and IEEE Computer Society. He received a BSME with High Honors from the University of Florida and an MSE from the University of California, Irvine.

Matt Schnaidt earned his professional engineering license while serving as the Battalion Adjutant. He received an MS from the Computer Science Department of the Naval Postgraduate School in 1998. Currently, Major Schnaidt is working on the Battle Management Command, Control and Communication component of the National Missile Defense Program.

H. J. Siegel is a Professor in the School of Electrical and Computer Engineering at Purdue University. He is an IEEE Fellow and an ACM Fellow. He received two BS degrees from MIT, and the MA, MSE, and PhD degrees from Princeton University. He has coauthored over 250 technical papers, was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and was an editor of the IEEE Transactions on Parallel and Distributed Systems.

Tracy D. Braun is a PhD student and Research Assistant at Purdue University. He received his BSEE with Honors and High Distinction from the University of Iowa in 1995. In 1997, he received his MSEE from the School of Electrical and Computer Engineering at Purdue. He is a member of IEEE, IEEE Computer Society, and Eta Kappa Nu honorary society. His research interests include parallel algorithms, heterogeneous computing, computer security, and software design.

Muthucumaru Maheswaran is an Assistant Professor in the Department of Computer Science at the University of Manitoba, Canada. He received a BSc degree from the University of Peradeniya, Sri Lanka and the MSEE and PhD degrees from Purdue University. He received a Fulbright scholarship to pursue

his MSEE degree at Purdue University. His research interests include computer architecture, distributed computing, heterogeneous computing, and resource management systems for metacomputing.

Shoukat Ali is an MSEE student at the School of Electrical and Computer Engineering at Purdue University. His main research topic is dynamic mapping of meta-tasks in heterogeneous computing systems. He has held teaching positions at Aitchison College and Keynesian Institute of Management and Sciences, both in Lahore, Pakistan. Shoukat received his BS degree from the University of Engineering and Technology, Lahore, Pakistan in 1996. His research interests include computer architecture, parallel computing, and heterogeneous computing.

Jong-Kook Kim is an MSEE student and Research Assistant in the school of Electrical and Computer Engineering at Purdue University, currently working on the DARPA/ISO sponsored BADD program. He received his BSEE from Korea University, Korea. He served in the ROK Army working with the US Army on the Theater Automated Command and Control Information Management System and received the US Army Commendation Medal. His research interests include heterogeneous computing and performance measures for distributed systems.

Cynthia E. Irvine is Director, Naval Postgraduate School Center for INFOSEC Studies and Research and an Assistant Professor of Computer Science at the Naval Postgraduate School. Dr. Irvine holds a PhD from Case Western Reserve University. She has over twelve years experience in computer security research and development. Her current research centers on architectural issues associated with applications for high assurance trusted systems, security architectures combining popular commercial and specialized multilevel components, and the design of multilevel secure operating systems.

Timothy Levin is currently doing research at the Naval Postgraduate School. He received a BS in Computer and Information Science from the University of California at Santa Cruz, 1981. His secure system work includes design of security features, and formal verification and formal covert channel analysis of an A1 operating system, and enterprise security features for a commercial relational database system. He has been certified by the NSA as a Vendor Security Analyst, for participation in their Trusted Product Evaluation Program.

Richard Freund is a founder and CEO of NOEMIX, a San Diego based startup to commercialize distributed computing technology. Freund is also one of the early pioneers in the field of distributed computing, in which he has written or co-authored a number of papers. In addition he is a founder of the Heterogeneous Computing Workshop, held each year in conjunction with IPPS/SPDP. Freund won a Meritorious Civilian Service Award during his career as a government scientist.

Matt Kussow, B.S.C.S., has 12 years of experience in software development, research, design, process analysis, and project management. Currently he holds the position of Vice President of Product Development at Noemix. He has extensive experience in designing and developing software for high performance computing, parallel algorithms, network computing, and database systems.

Michael Godfrey, B.S. C.I.S, UCSD Java Certified, has over 6 years of software research, design, and development experience with high performance computing, secure world wide web, health care, and data base systems for Science Applications International Corporation (SAIC) and Noemix, Inc. He has developed systems level applications on a variety of UNIX and Win32 platforms.

Alpay Duman is a LTJG in the Turkish Navy. He graduated from the Turkish Naval Academy with a BS in Operations Research with honors. He received his MSCS degree in the area of Systems Design and Architecture from the Naval Postgraduate School. He is currently a systems engineer at Turkish Navy Software Development Center working on a CORBA based communication infrastructure for Command Control Systems.

Paul F. Carff, LT US Navy, is a Masters student in the Computer Science Department at the Naval Postgraduate School, working on the Management System for Heterogenous Systems (MSHN). He received a BS in Engineering Physics from Santa Clara University in 1991. Prior to coming to Naval Postgraduate School, LT Carff served 3 years as a Nuclear Power Officer aboard the USS Salt Lake City (SSN 716).

Shirley Kidd is one of the supporting staff at the Heterogeneous Network & Computing Laboratory. She has 4 years of experience in the aerospace industry and another 6 years at a commercial marketing company during which she worked in both industries as a programmer analyst. She hasa BS in Applied Mathematics from the University of California, San Diego.

Viktor K. Prasanna (V.K. Prasanna Kumar) is a Professor in the Department of Electrical Engineering Systems, University of Southern California, Los Angeles. He obtained his PhD in Computer Science from Pennsylvania State University in 1983. He has published and consulted for industries in parallel computation, computer architecture, VLSI computations, and high performance computing for signal and image processing, and vision. He serves on the editorial boards of the Journal of Parallel and Distributed Computing and IEEE Transactions on Computers. He is the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing, and a Fellow of the IEEE.

Prashanth B. Bhat is a PhD candidate in Computer Engineering at the University of Southern California, Los Angeles. He received his B.Tech. degree in Computer Engineering from the Karnataka Regional Engineering College, India, in 1992. He received his ME degree in Computer Science and Engineering from the Indian Institute of Science, Bangalore, in 1994. His research interests include scheduling techniques for parallel and distributed systems, High Performance Computing and parallel computer architecture. During the summer of 1998, he was a research intern at Hewlett-Packard laboratories, Palo Alto.

Ammar Alhusaini is a PhD student in the Electrical Engineering Department at the University of Southern California. His main research interest is task scheduling in heterogeneous environments. He received an MS degree in computer engineering from the University of Southern California in 1996. He is a member of IEEE, IEEE Computer Society, and ACM.

QUIC: A Quality of Service Network Interface Layer for Communication in NOWs *

R. West, R. Krishnamurthy, W. K. Norton, K. Schwan, S. Yalamanchili, M. Rosu† and V. Sarat

Critical Systems Laboratory

Georgia Institute of Technology
Atlanta, GA 30332

Abstract

This project explores the development of a hardware/software infrastructure to enable the provision of quality of service (QoS) guarantees in high performance networks used to configure clusters of workstations/PCs. These networks of workstations (NOWs) have emerged as a viable high performance computational vehicle and are also being called upon to support access to multimedia datasets. Example applications include Web servers, video-on-demand servers, immersive environments, virtual meetings, multi-player 3-D games, interactive simulations, and collaborative design environments. Such applications must often share the interconnect with traditional compute intensive parallel/distributed applications that are usually driven by latency requirements in contrast to jitter, loss rate, or throughput requirements. The challenge is to develop a communication infrastructure that effectively manages the network resources to enable the diverse QoS requirements to be met. The major components of QUIC include (1) use of powerful, processors embedded in the network interfaces, (2) scheduling paradigms for concurrently satisfying distinct QoS requirements over multiple streams, (3) re-configurable hardware support to enable complex scheduling decisions to be made in the desired time frames, and (4) a flexible and extensible virtual communication machine that provides a uniform interface for dynamically adding hardware/software functionality to the network interfaces (NIs). This papers reviews the goals, approach and current status of this project.

*This work is supported in part by DARPA through the Honeywell Technology Center under contract numbers B09332478 and B09333218, the British Engineering and Physical Sciences Research Council with grant number 92600699, Intel Corporation, and the WindRiver Systems University Program.

†Now with IBM T. J. Watson Research Laboratories

1 Introduction

The continuing rapid decrease in the cost of both processor and network components has led to a tighter integration of computation and communication. The result has been an explosion of network-based applications characterized by the processing and delivery of continuous data streams and dynamic media [1, 6] in addition to servicing static data. Numerous examples can be drawn from web-based applications, interactive simulations, gaming, visualization, and collaborative design environments. The changing nature of the workload and cost/performance tradeoffs has prompted the development of a new generation of scalable media servers structured around networks of workstations interconnected by high speed system area network (SAN) fabrics. However the ability to construct scalable clusters that can serve both static and dynamic media is predicated on successfully addressing two major issues.

The first is that node architectures are based on a CPU-centric model optimized for uniprocessor or small-scale multiprocessor applications. This can lead to significant inefficiencies for distributed applications. Specifically, while CPU and wire bandwidths have been increasing rapidly over the years, memory and intra-node I/O bandwidths continue to improve at much slower rates, resulting in a performance gap that will continue to widen in the foreseeable future. This implies that interactions between the network and hosts utilizing main memory are expensive. Additional costs arise for such interactions from overheads due to I/O bus usage [4, 5], communication protocol implementations (e.g., if interrupts are used [2]), and interactions with the host CPU's memory management and caching infrastructure [3]. Consequently, network-based applications that produce, transport, and process large data sets suffer substantial losses in performance when these data sets must be moved through the memory and I/O hierarchies of multiple nodes.

0-7695-0107-9/99 $10.00 © 1999 IEEE

The second issue is the workload and performance characteristics of this new generation of network-based applications. Data types, processing requirements and performance metrics have changed placing new functional demands on the systems that serve them. Several key attributes are as follows [1, 6].

1. **Real-Time Response:** Interactivity and requirements on predictability, such as when to service video streams, make real-time response important. Such timing constraints cannot be met unless the network resources can be scheduled and allocated effectively.

2. **Shift from Quantitative to Qualitative Metrics:** With the new applications there has also been a shift in metrics that define their performance [1, 7]. Traditional quantitative metrics such as latency and bandwidth give way to more qualitative metrics such as jitter and real-time response. The smooth update of a video stream or the response time to a user access request is more important than minimizing transmission latency. The metrics clearly affects the choice of implementation techniques. For example packets may be dropped in a video or audio stream without compromising quality whereas this would be unacceptable for most transaction processing applications.

3. **Hardware Limitations:** The service and transfer of video streams and images places increasing demands on the memory and I/O bandwidths at a time when the bandwidth gap between the CPU and memory and I/O subsystems is growing. Furthermore the available physical bandwidth to the desktop and to the home will grow by several orders of magnitude. Wire bandwidths into the cluster that serves these machines with media will follow or lead this trend. This will exacerbate the "bandwidth gap" between the CPU and the wire, thereby making qualitative metrics more sensitive to the same.

4. **Heterogeneity:** Systems such as real-time media servers need to service hundreds and, possibly thousands, of clients typically each with their own *distinct* quality of service (QoS) requirements, such as packet dropping vs. reliable message transmission, low latency vs. jitter, or throughput vs. latency. We must concurrently meet diverse service requirements with the same set of hardware and software resources.

The QUIC project studies the issues inherent in Quality of Service management for cluster machines. Our project focuses on the functionality of the network messaging layer in providing QoS guarantees. Our approach is based on the development of an extensible QoS management infrastructure for which we carefully select components that are to be implemented within programmable network interfaces (NIs). In Section 2 we provide a brief overview of the project while a description of the overall Quality Management infrastructure can be found in Section 3. The rest of the paper describes the approach taken in the implementation of each of key QUIC components.

2 Project Overview and Goals

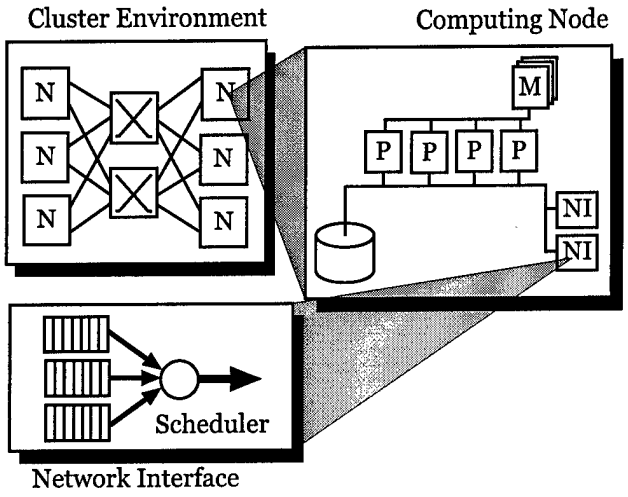


Figure 1: An Overview of the QUIC Development Infrastructure.

This project has a strong experimental component and therefore our infrastructure is biased towards rapid prototyping and evaluation. An overview of the QUIC infrastructure is illustrated in Figure 1. Our development environment is a cluster of 16 quad-Pentium Pro nodes interconnected via Intel's i960 based Intelligent I/O (I2O) network interface cards running the VxWorks operating system and interconnected via 100 Mbits/sec Ethernet. Concurrent development proceeds under the Windows NT and Solaris node operating system environments. Individual nodes have multiple network interfaces, multiple CPUs and eventually will coordinate services from multiple nodes[13].

Our goal is the development of an effective quality management infrastructure that can service a large number of connections, each with distinct service re-

quirements, while minimizing host-NI interactions for NIs whose functionality can be easily and dynamically extended. Our approach also investigates the extension of NI functionality to include computations on data streams as they pass through the network interface. By performing such stream computations 'on the fly' as data is passed through the interface we can avoid costly traversals of the memory hierarchy and thereby obtain finer control over service quality, for example real-time response. We are investigating supporting such stream computations through the use of programmable hardware in the form of dynamically configurable field programmable gate arrays (FPGAs) within the NIs. By placing quality management functionality "close" to the wire and I/O components attached to the I2O boards, we expect to enable QoS guarantees at higher levels of resource utilization than commodity clusters will otherwise permit. The following sections describe the individual aspects of the QUIC project.

3 QUIC Quality Management Infrastructure

The QUIC QoS infrastructure has several components that jointly permit the implementation of a variety of quality management functions and policies applied to the information streams into and out of CPUs.

- At the core of QUIC resides the extensible NI software architecture, which is designed for runtime extension with functions that manage the information streams used by particular applications. Two types of functions are supported: (1) *streamlets* that operate on the contents of data being streamed out of or into hosts, via the NIs on which the QUIC infrastructure's core components reside, and (2) *scheduling* or quality management functions that manage such streams, typically by applying certain scheduling algorithms to streams as they pass through NIs and the NI-host interface. This paper's principal focus is on these scheduling functions and on the manner in which they are applied to streams.

- QUIC offers two types of interfaces to applications: (1) a communication interface that directly supports its information streams, using standard communication protocols enhanced with the ability to specify scheduling functions or streamlets applied to them, and (2) an extension interface using which applications can define new quality management (scheduling) functions or streamlets to be applied to their information streams.

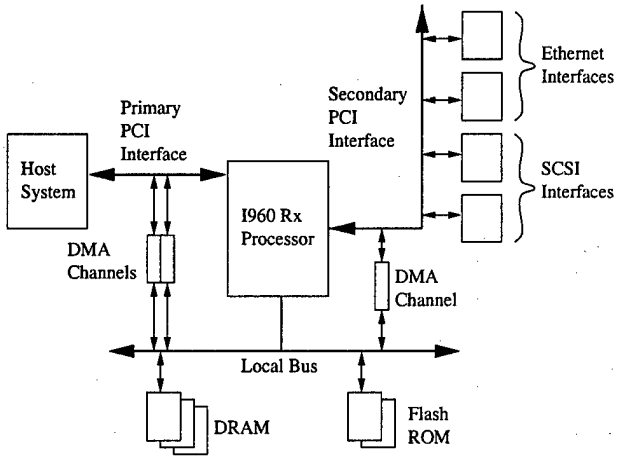


Figure 2: Architecture of the I2O Network Interface.

- QUIC also offers functions within each NI that permit the coordination among multiple NIs that jointly operate on certain information streams or cooperate to support applications. This 'control layer' of QUIC is also described below.

These functional components of QUIC are reviewed in the following sections.

3.1 Network Interface Architecture

Hardware Platform Intel's IQ80960RD66 Evaluation Platform Board serves as the underlying hardware upon which the QUIC NI software architecture is being instantiated. The board is designed to be a testbed for systems that offload I/O processing from the host. The architecture of the network interface is shown in Figure 2. The board resides in a PCI slot on the host machine and provides two network ports and two SCSI ports on an isolated PCI bus. The cards single processing unit is an Intel i960RX processor running at 66Mhz providing sufficient compute power to experiment with the movement of non-trivial computations to the interface[14, 15].

The quality management infrastructure executing within this NI will be layered on VxWorks, a real-time operating system from Wind River Systems. Our motivation for using VxWorks is its provision of cross compilation tools as well as a runtime layer using which our ideas concerning suitable NI functionality for quality management may be prototyped rapidly.

In general, the criteria for choice of an embedded kernel can be delineated very simply as follows: small footprint, lightweight, configurability, support for devices on the I/O controller card, and cross platform

development and debugging facilities. When choosing an NI for this project, the first option we considered was to construct a custom kernel for the i960. Our desire to rapidly prototype a functional system led to the choice of a commercial kernel for the i960 RD card. Two viable offerings from Wind River Systems are the IxWorks system developed as part of the I2O industry initiative and the VxWorks kernel. The ROM-resident IxWorks system provides a flexible environment for I2O device driver development, for monitoring driver performance, and it offers sophisticated message passing and event queues between multiple IOP boards. However, it assumes that message transport is performed using the I2O standard. We decided not to pursue this approach due to our focus on high performance (including low latency) communications. Toward this end, we wish to experiment with alternative I2O - host communication interfaces. Such experimentation is enabled by the second commercial offering from Wind River Systems for the i960 RD environment: the VxWorks development environment. The VxWorks operating system kernel is based on the same microkernel as the IxWorks I2O system. In contrast to IxWorks, VxWorks is highly configurable, as it can be scaled from a small footprint ROMable version to a larger footprint, full-featured Operating System. VxWorks makes no specific assumptions about the I2O - host interface[20, 19].

QUIC Communication Paradigm QUIC utilizes a general model of processing stream data closer to the network interface wherein the nature of the processing may include simple data computations as well as scheduling computations. Towards this end, QUIC enables the dynamic placement of computations within the NI. A request for the establishment of a connection will identify the required computations and specify desired levels of service quality based on which additional functionality such as admission control, policing (for network bandwidth) and scheduling may be performed. At this point, we simply point out that the establishment of a connection concerns not just the allocation of communication bandwidth to enable real-time transmission, but also the identification of the computations relevant to the data stream and the allocation of appropriate computational resources so that the data may be operated on in real-time. The two classes of computations considered in our work are (1) computations that operate on the actual stream data and (2) those that concern the runtime control (ie., scheduling) for streams. In either case, concerning communication coprocessors, this implies the runtime extension of these coprocessors with functionality suited for specific streams. An architecture and suitable interfaces for this purpose is being developed and is described as follows.

Virtual Communication Machine To application programs, the NI is abstracted as *virtual communication machine* (VCM), where its visible interface to applications is one that (1) defines the computations (instructions) it is able to perform on the applications' behalf and that (2) offers functions for the machine's runtime extension to add or subtract instructions as well as reporting necessary elements of its internal state.

Internally, the VCM's software architecture aims to provide adequate support for using the NI processor to improve the performance of streams used by network applications. Towards this end, the VCM provides an efficient environment for executing application-specific computational modules that can benefit from running 'closer to the network'. We use the term *application specific extension modules* (ASEM) to refer to such application-specific code that is 'directly' using the NI resources. In our testbed, the VCM will execute on the i960-based I2O coprocessor to which multiple disks and network links are attached. ASEMs provide a common abstraction and can be dynamically placed in the NI to process streams, perform scheduling, or manage NI resources such as disks.

Host applications communicate with the VCM through shared (host or NI) memory. The union of all these memory regions is called the VCM address space. Applications issue execution requests to the NI as VCM *tasks*. Tasks can be created and destroyed dynamically. The first VCM task of an application is created automatically when the application connects to the NI. The most common example of a task is one that executes a VCM program which in turn is comprised of core and 'added' *instructions*.

VCM programs are built as sequences of core and/or 'added' VCM instructions. Core instructions implement VCM functionality always resident on the VCM, including the functionality shared by all extensions and that required to configure the VCM's operation. Added instructions implement the extensions used by certain application programs. In other words, added instructions are functions specific to certain applications.

The VCM instruction dispatcher is implemented as an interpreter running on the NI processor. The dispatcher reads each VCM instruction, checks the availability of its parameters and activates the appropriate code. The NI processor transfers back results to the application by writing into the memory regions shared between the NI and the application.

The overhead of the interaction between applications and VCM instructions is low because of the shared-memory-based implementation. To further lower this overhead, access patterns to the shared regions are used to determine their placement in the host or the NI memory. Using shared memory also helps in decoupling the executions on the host and NI. Towards this end, applications and interface layer can build in the VCM address space arbitrary long chains of 'tasks-to-be-executed' and of 'tasks-completed' descriptors, respectively. In addition to the shared-memory-based interaction, the interface layer can also signal to application processes using signaling primitives provided in the host operating system. We note that certain elements in the proposed architecture were influenced by our experience with a small prototype built around an OC-3 ATM card (FORE SBA-200E). This commercial NI features a 25 MHz i960CA processors and 256K SRAM. Some details of our design appear next.

Extension Functionality and Interface We now comment on the 'extension' instructions part of the core instruction set. The conceptual basis for this work are (1) our previous experiences with the implementation of a VCM for FORE ATM interface boards[4] and (2) event-based mechanisms developed by our group for uniprocessor and distributed systems. The basic idea of these mechanisms is to permit applications to define events of certain types, to associate (at runtime) handlers for these events, and to create event channels to which event producers and consumers can subscribe. In our system, handlers are executed anytime an event is produced or consumed, at the producing and at the consuming side of the event channel. For online VCM extension, then, the application may produce an extension event and provide a new handler for this event type. The handler code is installed at runtime on the VCM, resulting in the creation of a new VCM instruction ready for use by the application program. Interactions of the new VCM instruction with lower level VCM facilities are resolved at installation time, as well.

The advantage of using this event-based approach is our ability to have any number of VCM's listen for extension events from any number of application programs, thereby offering a scalable approach to system extension even for large-scale machines. Some implementation details on VCM extension follow.

Each extension module is assigned one or more VCM instruction opcodes and control message IDs. An extension module is a collection of the following types of handlers:

- VCM instruction handlers invoked by the VCM instruction dispatcher upon encountering an instruction assigned to the module,
- control message handlers, invoked upon the delivery of a control message with an ID assigned to the module, and
- time-out handlers.

These handlers share the state of the extension modules, which is stored in the NI memory. We implemented five extension modules in the ATM prototype VCM. They all share the structure outlined above and they consist from 120 (the simplest) to 1170 (the most complex) lines of C code.

3.2 QUIC Runtime Environment

The QUIC runtime environment (RTE) is implemented as the VCM's runtime layer. This layer has two components, the first of which provides a set of technology-independent low-level communication abstractions. These abstractions make writing extension modules easier, as the application programmer does not have to be aware of the hardware details of a particular NI card. In addition, these abstractions are independent of the underlying networking technology (Ethernet, ATM, or Myrinet), thereby making the extension modules portable at the source code level. On top of this layer, the second component of the RTE is comprised of communication abstractions that support a collection of core services needed by most extension modules. By providing these services as part of the RTE, most of the functionality overlap between extension modules is eliminated. The implementation of this RTE component takes advantage of all the hardware support available on the NI to provide the best performance and, consequently, is highly dependent on the NI hardware.

Principles Guiding RTE Design The NI-dependent implementation of the first RTE component depends on the speed of the NI processor, the capacity of the NI memory, and the overheads of driving the network interconnect at full speed. A single-threaded library implementation of the RTE is the right choice for NIs with limited resources. An implementation based on a kernel for an embedded operating system should only be used for NIs with sufficient resources: fast processors, large memories, and interconnect-specific hardware that support the NI processor in driving the attached interconnects. Between these two alternatives sits the RTE implementation as a multithreaded

RTE Components The RTE is comprised of several components. The key issue in including components in the RTE is that they are used by several different extension modules. Our current design includes the following components.

Control Messaging System: This component implements reliable and in-order delivery of short messages between the NI processors. These short messages are called control messages and are used by extension modules on different NIs to exchange information. We found both reliability and in-order delivery very convenient when writing extension modules. Reliability needs to be implemented in the NIs as most SANs do not guarantee 100% message delivery. As message loss rates and latencies are relatively low on SANs, implementing in-order delivery should not increase the overhead of the NI processor considerably. Ideally, the latency of control messages should be only slightly higher than the hardware-imposed limit. To achieve this performance goal, buffers for control messages are preallocated in the NI memory. In our current prototype, reliable delivery is based on a sliding window protocol. Block acknowledgments are used between interface processors to acknowledge the receipt of multiple messages although upon request, certain control messages can be acknowledged immediately. This latter feature is useful in building fault-tolerant applications. For instance, we implemented this feature in our prototype and used it in an remote write extension module. This extension module is designed to improve the performance of applications that achieve fault-tolerance by maintaining a copy of their state in the memory of a remote host.

Time-out Components: A second RTE component implements several of time-out primitives, with different granularities and precision. The extended set of time-out primitives is necessary because we expect more precision and/or finer granularity to imply higher NI processor overhead.

Message Management Components: Routines for efficient assembling/disassembling of large messages from/into arbitrary collections of memory segments, placed either in the host or NI memory, are another RTE service. Our ATM-based prototype includes routines implementing zero-copy messaging: outgoing or incoming data is moved between the network registers and final destination without any intermediate copies in the host or NI memory. Our remote memory access and bulk messaging extension modules use these routines.

Memory Management Component: The RTE includes a dynamic memory management system. Our prototype includes a heap module which is implemented as buddy system to achieve better predictability.

4 QUIC Quality of Service Management

The QUIC scheduler represents an approach that emphasizes dynamic adaptation of scheduler parameters. We are pursuing an approach wherein existing scheduling algorithms can be modeled while permitting the algorithms to be adapted over time in an application-specific manner to respond to varying QoS needs.

4.1 QoS Management Paradigm

The QUIC project is exploring a flexible scheduling paradigm to concurrently satisfy diverse QoS requirements across multiple data streams. Packet priorities are dynamically updated at run-time to enable QoS requirements to be met. Essentially the packet priority is scaled as a function of the QoS that a packet has experienced up to that point in time *relative* to the QoS requested by the packet. Such an update operation has been referred to as priority *biasing* [7, 9, 8] since the priority value is biased by the relative degradation of its service. The biasing operation couples the effect of the scheduler (e.g., queuing delay) with the QoS demand (e.g., jitter bound). This distinguishes this approach from priority update mechanisms such as age counters that do not distinguish between QoS requested by distinct connections.

For example, consider only constant bit rate (CBR) connections where QoS is measured by the bandwidth allocated to a connection. The priority of a packet can be computed as the ratio of the queuing delay to the connection's inter-arrival time. Increasing queuing delay increases its priority in successive scheduling cycles. However, the rate at which the priority increases depends on the bandwidth of the connection. Such a priority biasing mechanism couples the ongoing effect of the switch scheduler (queuing delay) with a measure of the demands made by the application (connection bandwidth). As the negative impact of the switch scheduler grows so does the priority, effectively "biasing it" with time. Thus different connections are biased at different rates, i.e., higher speed connections are biased faster.

QUIC explores a hardware/software implementation of a generalized priority biasing framework. Our hypothesis is that by customizing biasing calculations by stream and data type and providing the ability to dynamically control biasing calculations we can achieve more effective utilization of network resources and

thereby satisfy the QoS requirements of a larger number of communication requests. Two major issues the framework must address are: *when* is the priority of a scheme biased and *how* is it biased. QUIC currently implements a dynamic window constrained scheduling algorithm (DWCS) that provides two parameters for controlling the "when" and "how" components of generalized priority biasing. This paradigm is quite powerful in that it has been shown to be able to model a range of existing scheduling algorithms [17] while it also provides for dynamic control of scheduling parameters thus providing new avenues for optimizing the performance of heterogeneous communication streams. The remainder of this section describes the current instantiation of DWCS hosted within the network interfaces.

4.2 QUIC Quality Management - The DWCS Approach

The first parameter utilized by DWCS controls the *interval* of time between priority adjustments of its second parameter. The second parameter is simply the *biasing value*, used to decide which stream has the highest priority and, hence, which stream should be scheduled for transmission. Simply, the biasing value is dynamically adjusted at specific intervals of time. Furthermore, DWCS is flexible, that the biasing value could be adjusted based on the needs of individual streams. Our current implementation utilizes packet *deadlines* and *loss-tolerance* as the two scheduling parameters. The motivation for this choice and detailed description are provided in the following.

Applications, such as real-time media servers need to service hundreds and, possibly, thousands of clients, each with their own quality of service (QoS) requirements. Many such clients can tolerate the loss of a certain fraction of the information requested from the server, resulting in little or no noticeable degradation in the client's perceived quality of service when the information is received and processed. Consequently, loss-rate is an important performance measure for the service quality to many clients of real-time media servers. We define the term *loss-rate*[16, 12] as the fraction of packets in a stream either discarded or serviced later than their delay constraints allow. However, from a client's point of view, loss-rate could be the fraction of packets either received late or not received at all.

One of the problems with using loss-rate as a performance metric is that it does not describe when losses are allowed to occur. For most loss-tolerant applications, there is usually a restriction on the number of *consecutive* packet losses that are acceptable. For example, losing a series of consecutive packets from an audio stream might result in the loss of a complete section of audio, rather than merely a reduction in the signal-to-noise ratio. A suitable performance measure in this case is a *windowed loss-rate*, i.e. loss-rate constrained over a finite range, or *window*, of consecutive packets. More precisely, an application might tolerate x packet losses for every y arrivals at the various service points across a network. Any service discipline attempting to meet these requirements must ensure that the number of violations to the loss-tolerance specification is minimized (if not zero) across the whole stream.

Some clients cannot tolerate any loss of information received from a server, but such clients often require delay bounds on the information. Consequently, these type of clients require deadlines which specify the maximum amount of time packets of information from the server can be delayed until they become invalid. Furthermore, some multimedia applications often require jitter, or delay variation, to be minimized. Such a requirement can be satisfied by restricting the service for an application to commence no earlier than a specified earliest time and no later than the deadline time.

To guarantee such diverse QoS requires fast and efficient scheduling support at the server. This section describes the features specific to a real-time packet scheduler resident on a server (specifically designed to run on either the host processor or the network interface card), designed to meet service constraints on information transferred across a network to many clients. Specifically, we describe Dynamic Window-Constrained Scheduling (DWCS), which is designed to meet the delay and loss constraints on packets from multiple streams with different performance objectives. In fact, DWCS is designed to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams.

4.3 The DWCS Scheduler

DWCS is designed to maximize network bandwidth usage in the presence of multiple packets each with their own service constraints. The algorithm requires two attributes per packet stream, as follows:

- *Deadline* – this is the latest time a packet can *commence* service. The deadline is determined from a specification of the maximum allowable time between servicing consecutive packets in the same stream (ie., the maximum inter-packet gap).
- *Loss-tolerance* – this is specified as a value x_i/y_i, where x_i is the number of packets that can be lost

or transmitted late for every *window*, y_i, of consecutive packet arrivals in the same stream, i. For every y_i packet arrivals in stream i, a minimum of $y_i - x_i$ packets must be scheduled on time, while at most x_i packets can miss their deadlines and be either dropped or transmitted late, depending on whether or not the attribute-based QoS for the stream allows some packets to be lost.

At any time, all packets in the same stream have the same loss-tolerance, while each successive packet in a stream has a deadline that is offset by a fixed amount from its predecessor. Using these attributes, DWCS: (1) can limit the number of late packets over finite numbers of consecutive packets in loss-tolerant or delay-constrained, heterogeneous traffic streams, (2) does not require a-priori knowledge of the worst-case loading from multiple streams to establish the necessary bandwidth allocations to meet per-stream delay and loss-constraints, (3) can safely drop late packets in lossy streams without unnecessarily transmitting them, thereby avoiding unnecessary bandwidth consumption, and (4) can exhibit both fairness and unfairness properties when necessary. In fact, DWCS can perform fair-bandwidth allocation, static priority (SP) and earliest-deadline first (EDF) scheduling.

4.4 DWCS Algorithm

Dynamic Window-Constrained Scheduling (DWCS) orders packets for transmission based on the *current* values of their loss-tolerances and deadlines. Precedence is given to the packet at the head of the stream with the lowest loss-tolerance. Packets in the same stream all have the same original and current loss-tolerances, and are scheduled in their order of arrival. Whenever a packet misses its deadline, the loss-tolerance for all packets in the same stream, s, is adjusted to reflect the increased importance of transmitting a packet from s. This approach avoids starving the service granted to a given packet stream, and attempts to increase the importance of servicing any packet in a stream likely to violate its original loss constraints. Conversely, any packet serviced before its deadline causes the loss-tolerance of other packets (yet to be serviced) in the same stream to be increased, thereby reducing their priority.

The loss-tolerance of a packet (and, hence, the corresponding stream) changes over time, depending on whether or not another (earlier) packet from the same stream has been scheduled for transmission by its deadline. If a packet cannot be scheduled by its deadline, it is either transmitted late (with adjusted loss-tolerance) or it is dropped and the deadline of the next packet in the stream is adjusted to compensate for the latest time it could be transmitted, assuming the dropped packet *was* transmitted as late as possible.

Pairwise Packet Ordering
Lowest loss-tolerance first
Same non-zero loss-tolerance, order EDF
Same non-zero loss-tolerance & deadlines, order lowest loss-numerator first
Zero loss-tolerance & denominators, order EDF
Zero loss-tolerance, order highest loss-denominator first
All other cases: first-come-first-serve

Table 1: Precedence amongst pairs of packets

Table 1 shows the rules for ordering pairs of packets in different streams. Recall that all packets in the same stream are queued in their order of arrival. If two packets have the same non-zero loss-tolerance, they are ordered earliest-deadline first (EDF) in the same queue. If two packets have the same non-zero loss-tolerance and deadline they are ordered lowest loss-numerator x_i first, where x_i/y_i is the current loss-tolerance for all packets in stream i. By ordering on the lowest loss-numerator, precedence is given to the packet in the stream with *tighter* loss constraints, since fewer consecutive packet losses can be tolerated. If two packets have zero loss-tolerance and their loss-denominators are both zero, they are ordered EDF, otherwise they are ordered highest loss-denominator first. If it is paramount that a stream never loses more packets than its loss-tolerance permits, then admission control must be used, to avoid accepting connections whose QoS constraints cannot be met due to existing connections' service constraints.

Every time a packet in stream i is transmitted, the loss-tolerance of i is adjusted. Likewise, other streams' loss-tolerances are adjusted *only if* any of the packets in those streams miss their deadlines as a result of queueing delay. Consequently, DWCS requires worst-case $O(n)$ time to select the next packet for service from those packets at the head of n distinct streams. However, the average case performance can be far better, because not all streams always need to have their loss-tolerances adjusted after a packet transmission.

Loss-Tolerance Adjustment. Loss-tolerances are adjusted by considering x_i/y_i, which is the original loss-tolerance for all packets in stream i, and x'_i/y'_i, which is the current loss-tolerance for all queued packets in stream i. The basic idea of these adjustments is to adjust loss numerators and denominators for all

buffered packets in the same stream i as the packet most recently transmitted before its deadline. The details of these adjustments appear in [17, 18]. Here, it suffices to say that DWCS has the ability to implement a number of real-time and non-real-time policies. Moreover, DWCS can act as a fair queueing, static priority and earliest-deadline first scheduling algorithm, as well as provide service for a mix of static and dynamic priority traffic streams.

4.5 Programmable Hardware Support

The explosive growth in the functionality of configurable or programmable hardware in the form of field programmable gate arrays (FPGAs) is changing the architecture of information systems that deal with data intensive computations [10, 11]. For example, we have seen the advent of configurable computing systems where programmable hardware is coupled with programmable processors. Hardware/software co-design has emerged as an associated design paradigm where the programmable hardware components (in the form of FPGAs) and software components (executed on processors) are designed concurrently with efficient trade-offs across the HW/SW boundary. While this paradigm is largely targeted towards embedded computer systems, we can apply relevant concepts to the design of intelligent network interfaces.

FPGA devices effectively represent hardware programmable alternatives to system level application specific integrated circuit (ASIC) designs and at a drastically reduced cost. Modern devices can be dynamically and incrementally re-programmed in microseconds, have increased memory on chip, and operate two to four times faster than current chips. FPGA devices perform particularly well on regular computations over large data sets. This technology naturally fits in architectures that stream and operate on large amounts of data as in the bulk of emerging network-based multimedia applications. The hardware functionality in the interfaces can be re-programmed "on the fly" almost as if we were swapping out custom devices.

We are motivated to include FPGA devices in the NIs for two specific reasons. The first is to host priority biasing calculations. Such calculations are inherently parallel with priorities being computed across many connections. A large number of relatively simple independent computations can be effectively supported within FPGAs. However, the overhead of such computations can render software NI implementations of certain biasing calculations either infeasible or greatly reduce the link utilizations that can be achieved. The second reason is the ability to host certain classes of data stream computations in the network interface. For example, data filtering, encryption, and compression are candidates for implementation with FPGAs available in the NI. Such computations can be naturally performed on data streams during transmission rather than via (relatively) expensive traversals through the memory hierarchy to the CPU. The VCM environment can provide access to these hardware devices via extension modules that can be used to load configuration data into the FPGAs. Thus, the abstractions used to extend the NI functionality dynamically are the same for functions implemented in software or programmable hardware. Our goal is to leverage this FPGA technology to enable more powerful yet (relatively) inexpensive network interfaces that can substantially enhance the performance of network applications.

5 Concluding Remarks

The goal of the QUIC project is the development of an effective quality management infrastructure that can service a large number of connections, each with distinct service requirements. Towards this end we are constructing an experimental infrastructure using a cluster of PCs interconnected by fast ethernet using i960 based interface cards. At the center of our efforts is an extensible software and quality management infrastructure. By placing quality management functionality "close" to the wire and I/O components attached to the I2O boards, we expect to enable QoS guarantees at higher levels of resource utilization than commodity clusters will otherwise permit. Our current efforts are geared towards creating a rapid prototyping environment to provide a basis for experimentation and investigation.

References

[1] K. Dienfendorff and P. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, vol. 30, no. 9, pp. 43-45, September 1997.

[2] Richard P. Martin and Amin M. Vahdat and David E. Culler and Thomas E. Anderson. Effects of Communication Latency, Overhead and Bandwidth in a Cluster Architecture. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[3] Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, Deborah A. Wallach, and William E.

Weihl. Efficient implementation of high-level languages on user-level communication architectures. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, 1995.

[4] Marcel-Catalin Rosu and Karsten Schwan and Richard Fujimoto, Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, August 1997.

[5] M. Rosu, K. Schwan, and R. Fujimoto. Supporting parallel applications on clusters of workstations: the virtual communication machine-based architecture. *Cluster Computing*, pp. 1029, November 1997.

[6] C. E. kozyrakis and D. Patterson. A new direction for computer architecture research. *IEEE Computer*, vol. 31, no. 11, pp. 24-32, November 1998.

[7] A. A. Chien and J. H. Kim. Approaches to quality of service in high performance networks. *Proceedings of the Workshop on Parallel Computer Routing and Communciation*, pp. 1-20, June 1997.

[8] J. H. Kim. Bandwidth and latency guarantees in low-cost high performance networks. Ph.D. Thesis, Department of Computer Sciences, University of Illinois, Urbana-Champaign, January 1997.

[9] D. Garcia and D. Watson. ServerNet II. *Proceedings of the Workshop on Parallel Computer Routing and Communication*, pp. 119-136, June 1997.

[10] J. Villasenor and W. H. Mangione-Smith. Configurable computing *Scientific American*, pp. 66-71, June 1997.

[11] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, H. Spaanenburg. Seeking solutions in configurable computing. *IEEE Computer*, vol. 30, no. 12, pp. 38-43, December 1997.

[12] Domenico Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 28(11):76–90, November 1990.

[13] *I_2O* Special Interest Group. www.i2osig.org/architecture/techback98.html.

[14] Intel. *i960 Rx I/O Microprocessor Developer's Manual*, April 1997.

[15] Intel. *IQ80960Rx Evaluation Platform Board Manual*, March 1997.

[16] Jon M. Peha and Fouad A. Tobagi. A cost-based scheduling algorithm to support integrated services. In *IEEE INFOCOMM'91*, pages 741–753. IEEE, 1991.

[17] Richard West and Karsten Schwan. Dynamic window-constrained scheduling for multimedia applications. Technical Report GIT-CC-98-18, Georgia Institute of Technology, 1998. To appear in the 6th International Conference on Multimedia Computing and Systems, ICMCS'99, Florence, Italy.

[18] Richard West, Karsten Schwan, and Christian Poellabauer. Scalable scheduling support for loss and delay constrained media streams. Technical Report GIT-CC-98-29, Georgia Institute of Technology, 1998.

[19] WindRiver Systems. *VxWorks Reference Manual*, 1 edition, February 1997.

[20] WindRiver Systems. *Writing I_2O Device Drivers in IxWorks*, 1 edition, September 1997.

Adaptive Distributed Applications on Heterogeneous Networks

Thomas Gross[1,2], Peter Steenkiste[1] and Jaspal Subhlok[3]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Departement Informatik
ETH Zürich
CH 8092 Zürich

[3] Department of Computer Science
University of Houston
Houston, TX 77204

Abstract

Distributed applications execute in environments that can include different network architectures as well as a range of compute platforms. Furthermore, these resources are shared by many users. Therefore these applications receive varying levels of service from the environment. Since the availability of resources in a networked environment often determines overall application performance, adaptivity is necessary for efficient execution and predictable response time. However, heterogeneous systems pose many challenges for adaptive applications. We discuss the range of situations that can benefit from adaptivity in the context of a set of system and environment parameters. Adaptive applications require information about the status of the execution environment and heterogeneous environments call for a portable system to provide such information. We discuss Remos (Resource Monitoring System), a system that allows applications to collect information about network and host conditions across different network architectures. Finally, we report our experience and performance results from a set of adaptive versions of Airshed *pollution modeling application executing on a networking testbed.*

1 Introduction

Many distributed applications have critical response time requirements. The timeliness of a response however depends on the availability of resources: network bandwidth to transfer information and processor cycles to perform computations. In heterogeneous environments, applications seldom have exclusive access to resources. Instead, network links and processors are shared by many applications and users.

The performance of a fast processor or network link can deteriorate to that of a slow one with additional computation load, but if the application can move to another system, then the user may not experience a slowdown. When running a distributed simulation, the impact of link congestion can be avoided by migrating to a different part of the network. A data warehouse may appear to stop operating when additional users start expensive queries, but if the data is replicated on another server, the application may switch to this server and thereby preserve the perception of a timely response. The transfer of a movie is subject to many dropped frames if there is network congestion. However, a smart filter may be able to remove non-essential frames from the movie and maintain audio and video synchronization by reducing bandwidth requirement.

All of these examples of adaptivity have been explored in various systems. In this paper we attempt to present a structure to these approaches that allows us to unify the development of interfaces between applications and environments. Since heterogeneous environments provide many challenges to application developers, it is important that the interface that provides network measurements is simple and portable. We believe that a uniform framework for developing adaptive applications and resource monitoring systems that work across different network architectures are the essential ingredients for speeding up the development of adaptive applications.

The remainder of this paper is organized as follows. We first describe the "space" of adaptation options that are available, using an example scientific simulation to illustrate the choices. We then give an overview of Remos system for collecting and reporting network status, and present performance results for an adaptive environmental modeling application. We conclude with a discussion of related work.

2 Adaptivity of applications

Adaptivity allows applications to run efficiently and predictably under a broader range of conditions. Support for adaptation may also allow applications to use less expensive service classes, e.g. best effort instead of guaranteed service. Some of the functionality (and complexity) associated with adaptation can be embedded in middleware, but we first have to understand the dimensions of adaptation before we can develop general purpose libraries or middle-

0-7695-0107-9/99 $10.00

ware layers to support adaptivity.

Applications can adapt along a number of "dimensions". In this paper we focus on the choice of resources (space dimension), the time of adaptation, and the interface between the application and the runtime system (or operating system, i.e., the system that is responsible for management of resources). In each case we first sketch the full spectrum of options available to applications in general, and we then focus on the options that are of most interest to distributed scientific simulations. We use Airshed environment modeling application described in Section 4 to illustrate the performance tradeoffs associated with adaptation.

2.1 Resource classes

Applications have access to a wide range of resources, and they often have a choice about how many and which resources they can use. An application can be adaptive with regard to the *number of processors* or nodes that it can use or its adaptivity may be restricted to the *space* of the network environment, i.e., the number of nodes is fixed but the identity of the nodes is determined dynamically.

Network resources are another candidate for adaptivity. Network bandwidth can sometimes be traded off with other parameters such as the fidelity of the data or the quality of the objects that are transferred. For example, by changing the size or the frame rate of a movie, an application can increase or decrease its bandwidth demands. Alternately, applications can make tradeoffs between different types of resources, e.g., compression can be used to reduce the bandwidth requirements, but then CPU cycles are required to compress and decompress the data.

Network resources are often not directly accessible to an application but their use is determined by the kind of service that the application requests. Recently, the networking community has been working on developing integrated services networks that can offer a range of services [4]. The *service class* dimension reflects the fact that an application can pick a service class that best matches its needs. This decision may (should) be based on dynamic conditions. E.g., when setting up a video conference over a network that supports differentiated service, the user or the application would like to pick the lowest service class (best effort service) that can provide sufficient bandwidth. A higher service class (e.g., expedited service) will be selected only if it can deliver the bandwidth that a lower service class is unable to do.

Scientific simulations can potentially use any of the above methods. The most common form of adaptation along the *space* dimension is likely to be the addition or deletion of execution nodes, as well as migration to a different subnet for execution. Rebalancing the load on different nodes and links can be used as a mechanism to adjust to the changing network status. Another option for adapting is modifying the mapping style of the computation onto the nodes, e.g., replication of data and computation to eliminate communication. In some cases application components can choose between multiple algorithms with different computation and communication requirements, and they can switch from one to the other when network conditions change. Finally, scientific simulations can also adapt in the *service class* dimension in a variety of ways, although relatively few networks today offer more than one service class.

2.2 Time of adaptation

Along the *time* dimension, at one extreme, applications adapt only at compile time. E.g., the user may hardwire the number of processors (nodes) into an application by specifying this number at the time the application is compiled. However, this scenario hardly qualifies as "adaptivity", so we will not discuss it further.

A more flexible option is that the program be compiled for a variable number of nodes and the actual number of nodes for execution is determined at the time the application is executed. Adaptation is in general based on the assumption that recent past conditions are a good predictor of near-term future conditions, an assumption that often holds. Dynamic adaptivity provides the most flexibility but also poses the biggest challenges to the application designer. The designer has two options with different benefit/complexity tradeoffs. One option is to limit adaptation to *load* or *start-up* time. This option is the easiest one since the applications has not set up any state yet. It has the obvious drawback that if conditions change during execution, the application will be unable to adapt to those changes. An example is an application that has a choice about what nodes, and thus what part of the environment, to use. A Web browser may be able to choose from several replicated servers or a proxy cache. An alternate model is to allow the application to adapt not only at startup but also at runtime. Such behavior is more complex to implement since it means that the application must be able to reconfigure itself. This capability requires changes in the application state and compute environment and therefore typically does not exist in today's applications; it must be added to make the application adaptive.

Runtime adaptation along the time dimension is addressed by protocols such as TCP and has also received the most attention from researchers studying network-aware applications. For applications that adapt dynamically, we can distinguish between applications that adapt periodically (e.g., a system that rebalances the loaded every k units of time) and systems that include *demand-* or *opportunity-driven* adaptivity. A system may adapt whenever some

performance parameter drops below a threshold or may opportunistically attempt to utilize extra resources as they become available.

Distributed simulations can benefit from adaptation both at startup and at runtime. At startup, they typically have to decide on the number of nodes to use [25] and on the setting of some control parameters, e.g., pipeline depth [19]. At runtime, they can periodically re-evaluate their options, and adaptation may take the form of migration of the executing program to a different part of the network or rebalancing or remapping of the computation on the executing nodes. This runtime adaptation adds considerable complexity to the program development and adaptation process, but is essential to get good performance for long running applications in dynamically changing conditions.

2.3 Information about the environment

To adapt, applications need information on the status of the environment. Traditionally, for network resources this task has been performed by communication protocols, such as TCP, so it is worthwhile to look at how these protocols collect information about network conditions. Protocols are often classified as using *implicit* or *explicit* feedback from the network. In protocols based on implicit feedback, the receiver monitors the incoming data stream and uses this stream to derive information about network conditions. TCP is a good example: dropped packets are viewed as a sign of congestion, and the sender responds by reducing its rate. In contrast, with explicit feedback, some entity inside the network provides explicit information about network conditions to senders. A good example is the ATM ABR traffic class: senders receive periodic information about network congestion conditions (e.g., congestion bit in rate management cells) or even the specific maximum traffic rate they are allowed to use (e.g., EPRCA).

Implicit feedback has the advantage that it does not require support from the network, so this approach to provide feedback is always feasible[1]. Implicit feedback also has some disadvantages: (i) it only allows incremental adaptation (i.e., when two hosts communicate, implicit information provides updates on how the bandwidth between these hosts evolves), and (ii) it is sometimes difficult to interpret the "information". (E.g., packet loss is an indication of congestion, but it is not always clear how the application should respond: pause for the duration of a round-trip time, reduce the congestion window, retransmit packets, etc.) Explicit feedback is in general easier to use, but it requires network support. Protocols today primarily rely on implicit feedback, and the same is true for most current network-aware applications. The reason is simple: implicit feedback does not require networking support, which does not exist.

Explicit information can be provided to the application in two ways. First, the network can provide feedback continuously. This approach is, e.g., employed for ABR traffic: a rate management cell is exchanged with the network for every 32 data cells. Continuous feedback is most often based on network properties that subsequently must be interpreted in the application space; for this reason this kind of coupling is also called *indirect*. An alternative is that applications receive a notification when specific events happen, e.g, an application receives an asynchronous notification when the network bandwidth drops below a certain threshold, or when the connection switches from one type of network to another [20]. Such notifications can be in the form of callbacks, or by invoking a specific event handler. With this style of interaction, the relationship between the network event (e.g., drop in bandwidth) and the actions (by the application or protocol software) is clearly established (e.g., when registering the handler). Therefore we call this style *event-driven* or direct coupling.

Scientific simulations can obtain network status information *externally* by using a tool to measure the activity on the network or *internally* by measuring the progress of work on different nodes and different parts of the network.

Simulations often use load balancing to improve the performance by giving less work to the nodes that are running slower than others. Load balancing can be implemented fairly well by internal measurements as the rate at which the work is progressing is a good indicator of the availability of resources. The general adaptation model, in which the application monitors its own performance and adapts when it observes a degradation (e.g. data loss), is widely applicable. It is possible to provide support for this form of adaptation through the use of frameworks [2] or other adaptation models [20].

However, other forms and dimensions of adaptation cannot be satisfied without external measurements. Selection of nodes at the the start of execution must be made with external measurements, as only those data points may be available at the time of invocation. Dynamic migration to a new set of nodes, as well as addition of nodes during execution, also requires external information, as internal information is limited to the current executing nodes.

Finally, it is useful to distinguish between the interface used by the application and the functionality supported by the network, since it is possible for a library or middleware layer to translate one interface into another. E.g., a library could translate continuous network feedback into event-based application feedback. A more interesting approach includes activities by the middleware: middleware could use a set of benchmarks to collect information on the conditions in a network (that does not provide explicit feedback) and present this information to the application in

[1] Implicit feedback typically makes some assumptions about the network, e.g., it considers packet loss to be a sign of congestion.

an explicit form. In this paper, we focus on the application level interface, and we only touch briefly on the lower level interface when we discuss implementation options.

2.4 Network-application interactions

To be able to adapt along all three dimensions, applications will need information on network conditions, which span a matching space with the same dimensions. How applications collect network information determines how easily applications can explore this space.

Implicit information is based on experience, which severely restricts what part of the information space is explored: the application only learns about the part of the space it currently operates in. This means that it can collect information only on the network conditions along the paths it is currently using and on the service class it is in. Implicit feedback provides information along the time dimension, but only while the application is actively using the network. At startup or after the application has been idle for a while, no useful information is available.

We argue that given the limits on what information can be collected using implicit feedback, mechanisms must be provided so that applications can get explicit information on network conditions, e.g., by querying a standard interface. Such an interface should allow applications to collect information on network conditions in the entire network space (space, time and service class dimensions), allowing applications to make adaptation decisions in space, across service classes, and at startup.

One can argue that the restrictions on the type of information that can be collected using implicit feedback is not fundamental. Applications can use probing to explore the entire information space, e.g., they can periodically try all service classes and they can measure the network performance between every pair of usable hosts. While this approach may be appropriate in some cases, it is in general undesirable. First, developing effective network probing routines is difficult; it is not something that application developers should be required to do. Second, probing can be expensive, both in terms of elapsed time for the application and consumed network resources. Furthermore, excessive probing may disturb the measurements taken by this and other applications. In fact, large scale probing by applications would negate many of the advantages of implicit feedback. If probing is needed, it should be performed by a middleware layer. Then the probing code can be developed as part of the network architecture, and the collected information can be shared by many applications.

Note that we are proposing explicit feedback as a complementary mechanism to implicit feedback, and not as a replacement. Implicit feedback has clear advantages when used appropriately. Implicit feedback will remain useful as an inexpensive way of getting continuous feedback, once a particular operating point along the space and service dimensions has been selected. Implicit feedback is also likely to give more accurate and timely information (in a narrower part of the operating space) than explicit feedback.

3 A base system: Remos

The Remos API provides a query-based interface that allows clients to obtain "best-effort" information [14] on network conditions. The applications specifies the kind of information it needs, and Remos supplies the best available information. To limit the scope of the query, the application must select network parameters and parts of a larger network that are of interest. In this section we briefly describe the main Remos features. A more detailed description can be found elsewhere [14].

3.1 Level of abstraction

To accommodate the diverse application needs, the Remos API provides two levels of abstraction: high level flow-based queries and lower level topology-based queries.

Remos supports flow-based queries. A *flow* is an application-level connection between a pair of computation nodes. Queries about bandwidth and latency of sets of flows form the core of the Remos interface. Using flows instead of physical links provides a high level of abstraction that makes the interface portable and independent of system details. Flow-based queries place the burden of translating network-specific information into application-oriented information on the implementor of the API. However, flows are an intuitive abstraction for application developers, and they allow the development of adaptive network applications that are independent of the heterogeneity inherent in a network computing environment.

Remos also supports queries about the network *topology*. The reason we expose a network-level view of connectivity is that certain types of questions are more easily or more efficiently answered based on topology information. E.g., finding the pair of nodes with the highest bandwidth connectivity is expensive using only flow-based queries. The topology information provided by Remos consists of a graph with compute nodes, network nodes, and links, each annotated with their physical characteristics, such as latency and available bandwidth. Topology queries return a *logical* interconnection topology. This means that the graph represents the network behavior as seen by the application, and does not necessarily reflect the physical topology. Using a logical topology gives Remos the option of hiding network features that do not affect the application. E.g., subnets can be replaced by (logical) links if their internal

structure does not affect applications. Topology information is in general harder to use than flow-based information, since the complexity of translating network-level data into application-level information is mostly left to the user.

3.2 Dynamic resource sharing

Since networks are a shared resource, it is important to account for the manner in which resources are shared by multiple flows. Since multi-party applications use multiple flows, it is not only necessary to account for sharing across applications, but also across flows belonging to the same application. To be able to consider the effects of "internal" sharing, Remos supports multi-flow queries in which the application lists all its flows simultaneously. Applications can generate flows with very diverse sharing characteristics, ranging from constrained low-bandwidth audio to bursty high-bandwidth data flows. Remos collapses this broad spectrum into three types of flows. *Fixed flows* have a specific bandwidth requirement. *Variable flows* have related requirements and demand the maximum available bandwidth that can be provided to all such flows in a given ratio. (E.g., all flows in a typical all-to-all communication operation have the same requirements.) Finally, *independent flows* simply want maximum available bandwidth. These flow types also reflect priorities when sufficient resources are not available to satisfy all the flows. *Fixed flows* are considered first, followed by *variable flows*, then *independent flows*.

Determining how the throughput of a flow is affected by other messages in transit is very complicated and network specific. Remos approximates this complex behavior by assuming that, all else being equal, the bottleneck link bandwidth is shared equally by all flows (that are not bottlenecked elsewhere). If other information is available, Remos can use different sharing policies when estimating flow bandwidths. The basic sharing policy assumed by Remos corresponds to the max-min fair share policy [11]. Applications that use topology-based queries are themselves responsible for taking the effects of both internal and external sharing into account.

3.3 Accuracy

Applications ideally want information about the level of service they can expect to receive in the future, but most users today must use past performance as a predictor of the future. Different applications are also interested in activities on different timescales. A synchronous parallel application expects to transfer bursts of data in short periods of time, while a long running data intensive application may be interested in throughput over an extended period of time. For this reason, relevant queries in the Remos interface accept a timeframe parameter that allows the user to request data collected and averaged for a specific time window.

Network information such as available bandwidth changes continuously due to sharing and as a result, characterizing these metrics by a single number can be misleading. E.g., knowing that the bandwidth availability has been very stable represents a different scenario from it being an average of rapidly changing instantaneous bandwidths. To address these aspects, the Remos interface adds statistical variability and estimation accuracy parameters to all dynamic quantitative information. Since the actual distributions for the measured quantities are generally not known, we present the variability of network parameters using quartiles [12].

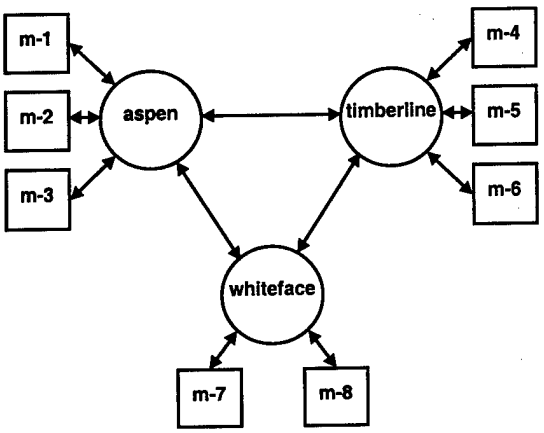
3.4 Implementation

An initial version of Remos API has been implemented. It has two components, a *collector* and *modeler*, that are responsible for network-oriented and application-oriented functionality, respectively. The collector is responsible for collecting low-level network information. Such data can be collected in many ways, e.g., one can periodically run benchmarks that probe the network for available bandwidth or rely on information gathered by applications [21]. Our current implementation uses a third method: the collector explicitly queries routers using SNMP [3] for both topology and dynamic bandwidth information. The use of SNMP to obtain information about the state of a network is a standard way of monitoring networks, and it should allow us to collect detailed information in a relatively non-intrusive way on a broad set of networks. The modeler is a library that is linked with the application; it translates the information provided by the collector into a logical topology graph or per-flow data in response to application requests. The modeler-collector architecture is in part motivated by the need to support scalability and network heterogeneity. In large networked environments, multiple collectors may have to be deployed, and each collector can collect information in a way that is most appropriate for the network it is responsible for. Work is in progress on implementing collectors that use sources of network information other than SNMP, e.g., by active measurements.

For the results presented in this paper we used the Remos interface on a dedicated IP-based testbed at Carnegie Mellon University that is illustrated in Figure 2.

	Explicit	Implicit
Indirect (continuous)	Queries to network management database Rate management cells	Counting lost packets
Event-driven (direct)	Handlers to react to changes in the network	Reacting to lost retransmissions

Figure 1: Examples for two dimensions of application/network coupling.



Links: 100Mbps point-to-point ethernet
Endpoints: DEC Alpha Systems (*manchester-** labeled *m-**)
Routers: Pentium Pro PCs running NetBSD *(aspen,timberline, whiteface)*

Figure 2: Testbed used for Airshed experiments

4 Case study in adaptive execution: Airshed pollution modeling

We have developed a suite of tools to develop adaptive distributed programs driven by Remos and have gained experience with programs ranging from small kernels like fast Fourier transforms to complete applications like Airshed pollution modeling and magnetic resonance imaging[5]. In this paper, we focus exclusively on Airshed pollution modeling application and present results comparing the performance of the basic implementation with various adaptive versions.

The Airshed application [15] models formation, reaction, and transport of atmospheric pollutants and related chemical species. We implemented a distributed version of Airshed using Fx data parallelism [8]. Data parallelism in Fx is similar to High Performance Fortran [9], so these observations apply to other applications as well. An adaptive version of Airshed was developed using integrated task and data parallelism in Fx [24]. For efficient execution, this application involves significant communication in the form of array redistributions since the various chemistry and transport phases access the main particle array along different dimensions. The details of the Airshed implementation are described in [23].

We executed Airshed using a tool that automatically selects the best nodes for execution based on the network information provided by Remos. The details of this node selection procedure and its validation is discussed in [22]. Table1 presents the results obtained on our networking testbed, which consists of a number of DEC Alpha workstations connected via three routers. The testbed allows us to configure the bandwidth between the routers, as well as to apply various traffic patterns for controlled experiments. Figure 2 shows the set-up.

We observe that automatic node selection has little impact on performance in the absence of network traffic. In the presence of a fixed traffic stream that saturates one of the communication links, automatic node selection more than halves the execution time. The reason is that Airshed is a SPMD application with a significant communication component, and saturation of a single link creates a bottleneck that slows down the entire computation. However, it is possible to select a set of nodes automatically using Remos information such that the busy links are avoided for program communication. The last two columns in the table show performance on the network with load generators that simulate moderate utilization of network resources. We observe that the performance is considerably enhanced with automatic node selection as the node selection succeeds in avoiding congested links and busy processors in many cases. However, such enhancements are not always possible when the network is heavily used, and hence the performance advantage is not to the extent observed for a single congested link.

The results highlight the importance of simple adaptation in the *resource space* dimension at start-up time, and demonstrate that a toolset based on external measurements can effectively drive such adaptation. Note that internal measurements made by a program are not of any use in deciding which nodes the application should be started on. The main drawback of adaptation only at start-up time is that network conditions change over time, and hence adaptation during execution is important for long running applications.

Execution Node Selection	Execution time with external load and traffic			
	No Network Traffic	Fixed Network Traffic	Dynamically varying Traffic	Dynamically varying Traffic and Load
Random	652	1726	1125	2121
Automatic	650	674	754	1420

Table 1: Performance results of 5-node Airshed in different network conditions. Execution times using automatic node selection are compared with those obtained with random node selection. For the case of dynamically varying traffic, only 1/4 of the Airshed processing was done in one invocation and the results shown are scaled up for comparison

Table2 presents preliminary results from a dynamic adaptive version of Airshed. This version queries Remos for network status after every major simulation step, and migrates to a new set of nodes if the current set of nodes or links become considerably more busy than other parts of the network. For the purpose of comparison, we created an adaptive version that would not actually migrate (but had adaptation support built into it) and compared it to the actual migrating adaptive application, under different network conditions. Both were started on the same set of nodes, and a fixed traffic pattern was maintained for the duration of the experiment. The interfering and non-interfering patterns are relative to the set of nodes and links on which the application was started.

We first observe that both the versions run slower than the non-adaptive versions of Airshed discussed earlier, even in the absence of any load or traffic. This observation points out the fixed overhead of adaptation support. In the absence of interfering network traffic, the migrating version executes slightly slower than the static version. This difference reflects two types of overheads associated with migration. First, there is a cost associated with analyzing and deciding the best nodes for execution. Second is the cost associated with unnecessary migration, which can happen because of the heuristic nature of adaptation decisions. Finally, we observe that the adaptive version performs significantly better in the presence of interfering traffic. The general conclusion is that support for adaptation entails moderate overheads, but it can minimize the impact of external traffic on execution times.

This experiment highlights the importance of runtime adaptation and demonstrates that an external tool like Remos is effective in driving the dynamic adaptation process. Note that runtime migration cannot be done with internal application measurements, since they are not available for the nodes that the application is not currently executing on. However, internal measurements can be used for load balancing, which is an example of an application modifying its demands in response to changes in the resource availability. It is clear that adaptation by migration exploits a degree of freedom in the resource dimension that is not available to load balancers.

5 Related work

An important contribution of our research is to provide a structure to adaptivity options, especially in the context of distributed simulations. We are not aware of any work that specifically addresses this aspect. However, a number of projects address measurement and management of network resources that complement the Remos system discussed in the paper. We also discuss some other approaches to adaptivity reported in the literature.

5.1 Network resource management and measurements

A number of resource management systems allow applications to make queries about the availability of computation resources, some examples being Condor [13] and LSF (Load Sharing Facility). Resource management systems for large scale internet-wide computing is an important area of current research, and some well known efforts are Globus [6] and Legion [7]. These systems provide support for a wide range of functions such as resource location and reservation, authentication, and remote process creation mechanisms. Recent systems that focus on measurements of communication resources across internet wide networks include Network Weather Service (NWS) [26] and topology-d [16]. NWS makes resource measurements to predict future resource availability, while topology-d computes the logical topology of a set of internet nodes. Both these systems actively send messages to make communication measurements between pairs of computation nodes. A number of sites are collecting Internet traffic statistics, e.g., [1]. This information is not in a form that is usable for applications, and it is typically also at a coarser grain than what applications are interested in using. Another class of related research is the collection and use of application specific performance data, e.g., a Web browser that collects information about the response times of different sites [21]. Related work also addresses estimating stochastic values [18] that represent varying quantities on networks.

In comparing with some of these projects, the Remos interface focuses on providing good abstractions and sup-

Execution Node Set	Execution time with traffic patterns (seconds)			
	No Traffic	Non-interfering Traffic	Interfering Traffic-1	Interfering Traffic-2
Fixed	862	866	1680	1826
Adaptive	941	974	1045	955

Table 2: Execution times of adaptive version of Airshed executing on a fixed set of nodes and on dynamically selected nodes

port for application level access to network status information and allows for a closer coupling of applications and networks. Remos implementations make measurements at network level when possible; this strategy minimizes the measurement overhead and yields key information for managing sharing of resources.

5.2 Other models and extensions

Several approaches to provide adaptivity without changes to the programming model have been researched in the literature. Nevertheless, it is interesting to note that these systems include components that map directly into the concepts discussed in this paper.

A number of groups have looked at the benefits of explicit feedback to simplify and speed up adaptation (e.g., [10]). However, the interfaces developed by these efforts have been designed specifically for the scenarios being studied.

The *Quality Objects* QuO system[27] provides adaptivity in the context of object-oriented programming. To provide the feedback between applications and environment, the QuO system includes *system condition objects* that drive adaptivity either implicitly or explicitly.

An adaptive system that provides a shared-memory programming model for a network of workstations or PCs can take advantage of additional nodes and also deal with withdrawal of a nodes [17]. Here the control of adaptivity rests with the (software) distributed shared-memory system, but the application (or the compiler) determines the points in the execution of the program where adaptivity is possible.

6 Concluding remarks

Figure 1 gives examples of the 4 different kinds of couplings between applications and networks that are discussed in this paper. Network-aware applications today focus overwhelmingly on implicit interaction. We argue that this state of affairs is due to the current (lack of) support for other interactions models by network architectures. As network architectures begin to provide information that is more accurate, more timely, and more detailed, network-aware applications will be motivated to also explore explicit interaction.

We show how the adaptivity options for distributed simulations fit in the general framework of adaptive execution. The results demonstrate that portable external mechanisms for network measurements are necessary to support effective adaptive execution of large distributed scientific applications.

Acknowledgments

We appreciate contributions and comments by D. Bakken, J. Bolliger, P. Dinda, B. Lowekamp, D. O'Hallaron, N. Miller, D. Sutherland, and J. Zinky.

Effort sponsored by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] http://www.nlanr.net.

[2] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Trans. Softw. Eng.*, 24(5):376 – 390, May 1998.

[3] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2), January 1999. RFC 1905.

[4] Dave Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanisms. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 14–26, Baltimore, August 1992. ACM.

[5] W. Eddy, M. Fitzgerald, C. Genovese, A. Mockus, and D. Noll. Functional image analysis software - computational olio. In A. Prat, editor, *Proceedings in Computational Statistics*, pages 39–49, Heidelberg, 1996.

[6] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[7] A. Grimshaw, W. Wulf, and Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[8] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 2(3):16–26, Fall 1994.

[9] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, December 1996.

[10] J. Inouye, S. Cen, C. Pu, and J. Walpole. System support for mobile multimedia applications. In *Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 143–154, St. Louis, May 1997.

[11] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, July 1981.

[12] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[14] Bruce Lowekamp, Nancy Miller, Dean Sutherland, Thomas Gross, Peter Steenkiste, and Jaspal Subhlok. A Resource Query Interface for Network-Aware Applications. In *7th IEEE Symposium on High-Performance Distributed Computing*, pages 189–196, Chicago, July 1998.

[15] G. McRae, A. Russell, and R. Harley. *CIT Photochemical Airshed Model - Systems Manual*. Carnegie Mellon University, Pittsburgh, PA, and California Institute of Technology, Pasadena, CA, February 1992.

[16] K. Obraczka and G. Gheorghiu. The performance of a service for network-aware applications. Technical Report TR 97-660, Computer Science Department, University of Southern California, Oct 1997.

[17] A. Scherer, H. Lu, T. Gross, and W. Zwaenepoel. Transparent adaptive parallelism on nows using openmp. In *Proc. 7thACM Symp. on Principles and Practice of Parallel Prog. (PPoPP'99)*, page (to appear), Atlanta, GA, May 1999. ACM.

[18] J. Schopf and F. Berman. Performance prediction in production environments. In *12th International Parallel Processing Symposium*, pages 647–653, Orlando, FL, April 1998.

[19] Bruce Siegell and Peter Steenkiste. Automatic selection of load balancing parameters using compile-time and run-time information. *Concurrency - Practice and Experience*, 9(3):275–317, 1996.

[20] Peter Steenkiste. Adaptation models for network-aware distributed computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99)*, Orlando, January 1999. IEEE. Springer-Verlag.

[21] M. Stemm, S. Seshan, and R. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, June 1997.

[22] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[23] J. Subhlok, P. Steenkiste, J. Stichnoth, and P. Lieu. Airshed pollution modeling: A case study in application development in an HPF environment. In *12th International Parallel Processing Symposium*, pages 701–710, Orlando, FL, April 1998.

[24] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, June 1997.

[25] Hongsuda Tangmunarunkit and Peter Steenkiste. Network-aware distributed computing: A case study. In *Second Workshop on Runtime Systems for Parallel Programming (RTSPP)*, page Proceedings to be published by Springer, Orlando, March 1998. IEEE. Held in conjunction with IPPS '98.

[26] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. Technical Report TR-CS97-540, University of California, San Diego, May 1997.

[27] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.

7 Biographies

Thomas R. Gross is a faculty member in the School of Computer Science at Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA. (thomas.gross@cs.cmu.edu). He joined CMU in 1984 after receiving a Ph.D. in Electrical Engineering from Stanford University. He also has an appointment at ETH Zurich. He is interested in tools, techniques, and abstractions for software construction and has worked on many aspects of the design and implementation of programs. To add some realism to his research, he has focussed on compilers for uni-processors and parallel systems. He has worked on many areas of compilation (code generation, optimization, debugging, partitioning of computations, data parallelism and task parallelism) and software construction (frameworks, patterns, components). In his current research, Thomas Gross and his colleagues investigate network- and system-aware programs – i.e. programs that can adjust their resource demands in response to resource availability.

Further information (including additional references and downloadable versions of many papers) can be found at www.cs.cmu.edu/ cmcl

Peter Steenkiste received the degree of Electrical Engineer from the University of Gent in Belgium in 1982, and the MS and PhD degrees in Electrical Engineering from Stanford University in 1983 and 1987, respectively. He then joined the School of Computer Science at Carnegie Mellon University, where he is current a Senior Research Scientist.

Peter Steenkiste's research interests are in the areas of networking and distributed computing. While at CMU, Peter Steenkiste worked on a number of projects in the high-performance networking and distributed computing area. Earlier projects include Nectar, the first workstation clusters built around a high-performance, switch-based local area network, Gigabit Nectar, a heterogeneous multi-computer, and Credit Net, a high-speed ATM network. Peter Steenkiste is currently exploring the notion of "application-aware networks", i.e. networks that can delivery high quality, customized services to applications, in the context of the Darwin project.

He is also involved in the Remulac project, which is developing middleware in support of network-aware applications. More information can be found on his web page http://www.cs.cmu.edu/ prs

Peter Steenkiste is a member of the IEEE Computer Society and the ACM. He has been on a number of program committees and is an associated editor for IEEE Transactions on Parallel and Distributed Systems.

Jaspal Subhlok received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India in 1984, and Ph.D. degree in computer science from Rice University, Houston in 1990. Between 1990 and 1998 he served as a member of the research faculty in the School of Computer Science at Carnegie Mellon University, Pittsburgh. He is currently an Associate Professor of Computer Science at the University of Houston.

Dr Subhlok's technical areas of interest are compilers, tools and runtime systems, particularly in the context of parallel and distributed computing. His research involves design of algorithms and systems to solve a variety of problems in programming and runtime support for parallel and networked systems. The focus of his current research is on "network aware" distributed computing, that spans the development of tools and frameworks to support applications that can dynamically adapt to changing system resources. His earlier projects included development and standardization of integrated task and data parallelism in the context of High Performance Fortran, algorithms and tools for automatic mapping of mixed task and data parallel programs, and validation of job scheduling strategies with actual supercomputer workloads.

Further information is available from http://www.cs.uh.edu/ jaspal.

Author Index

— *Notes* —

— *Notes* —

Press Activities Board

IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the Online Catalog, *http://computer.org*, for a list of products.

IEEE Computer Society Proceedings

The IEEE Computer Society also produces and actively promotes the proceedings of more than 141 acclaimed international conferences each year in multimedia formats that include hard and softcover books, CD-ROMs, videos, and on-line publications.

For information on the IEEE Computer Society proceedings, send e-mail to cs.books@computer.org or write to Proceedings, IEEE Computer Society, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1 714-821-8380. FAX +1 714-761-1784.

Additional information regarding the Computer Society, conferences and proceedings, CD-ROMs, videos, and books can also be accessed from our web site at *http://computer.org/cspress*

1/29/99